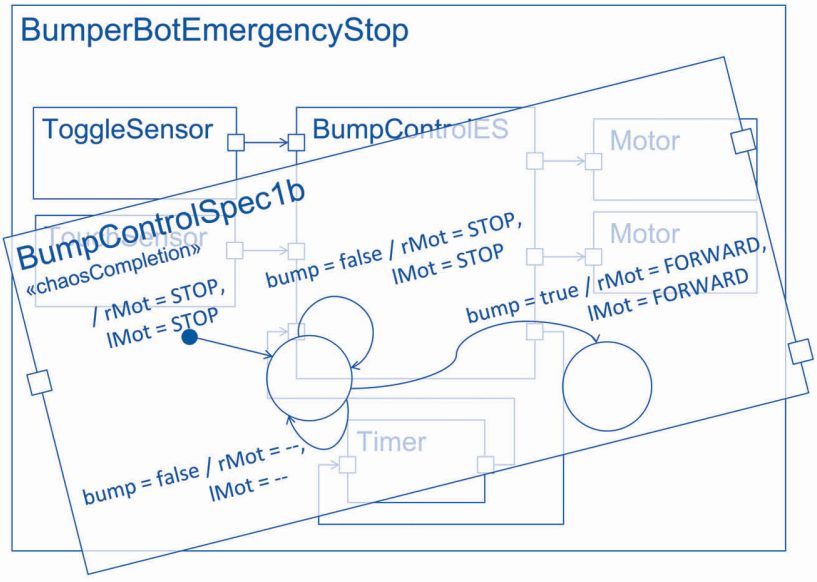
RWTH AACHEN

SE SOFTWARE ENGINEERING

Jan O. Ringert

Analysis and Synthesis of Interactive Component and Connector Systems



Aachener Informatik-Berichte,
Software Engineering

Band 19

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Analysis and Synthesis of Interactive Component and Connector Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Jan Oliver Ringert
aus Hildesheim

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
University Professor Marsha Chechik, PhD

Tag der mündlichen Prüfung: 24.04.2014

[Rin14] J. O. Ringert:
Analysis and Synthesis of Interactive Component and Connector Systems.
Shaker Verlag, ISBN 978-3-8440-3120-1. Aachener Informatik-Berichte, Software Engineering, Band 19. 2014.
www.se-rwth.de/publications/

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 19

Jan Oliver Ringert

Analysis and Synthesis of Interactive Component and Connector Systems

Shaker Verlag
Aachen 2014

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2014)

Printed in Germany.

ISBN 978-3-8440-3120-1
ISSN 1869-9170

Shaker Verlag GmbH • P.O. BOX 101818 • D-52018 Aachen
Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9
Internet: www.shaker.de • e-mail: info@shaker.de

Abstract

The development of distributed interactive hard- and software systems is a challenging endeavor. Component and connector (C&C) architecture descriptions address the complexity of interactive systems by formalizing the logical and physical decomposition of systems into subsystems. C&C descriptions model components with well-defined interfaces and component interaction via connectors.

Current modeling languages and specification mechanisms rely on the traditional, implementation-oriented hierarchical decomposition of systems into subsystems. We are developing modeling languages and methods that crosscut these boundaries and allow to capture the partial knowledge available to different stakeholders involved in a system's design. Usage scenarios include formalizing crosscutting knowledge about the C&C structure of the system, specifying behavior and interaction, and providing a model-based implementation for code generation and deployment.

We present a language to describe design decisions and knowledge available about a system's decomposition in partial C&C views. This language is based on C&C modeling languages and adds powerful abstraction mechanisms for hierarchical containment, connectedness, and interfaces. C&C views may specify valid, invalid, alternative, and dependent designs. Our analysis methods can verify whether a C&C model satisfies a C&C view. We have also developed a synthesis method to automatically compute a satisfying C&C model for a given specification, if one exists.

To describe the interaction behavior of components, we have developed a modeling language for automata that are embedded in components and interact by sending and receiving messages via the components' typed input and output ports. This language has various mechanisms for underspecification of component behavior. We have developed tool support to verify the implementation of components and component compositions against underspecified models. The verification enables incremental development of component behavior based on stepwise refinement. We also present a code generation framework for the educational Lego NXT robotics platform to demonstrate an application of our work in the robotics domain.

Prototype implementations and evaluation in a user study, a case study, and over example systems show promising results towards a comprehensive model-based development environment for interactive component and connector systems.

Acknowledgments

I would like to express my deep gratitude to Prof. Dr. Bernhard Rumpe for giving me the opportunity to realize this thesis, for composing a strong research group fun to work with, and for the many fruitful discussions and advice on the research presented here.

I would like to thank Prof. Marsha Chechik, PhD for being the second examiner of my thesis and for her helpful comments on improving the presentation of my research. My thanks also goes to Prof. Dr.-Ing. Stefan Kowalewski for heading my PhD exam committee and to Prof. Dr. Thomas Noll for being part of the committee as well as for many interesting discussions on my research in previous years.

I would also like to thank the professors of the DFG research training group "Algorithmic synthesis of reactive and discrete-continuous systems" (AlgoSyn) for providing a lively and productive forum for PhD students and researchers to collaborate and exchange their ideas on a common research topic. Special appreciation goes to Prof. Dr. Dr. h.c. mult. Wolfgang Thomas for the great organization and his exceptional care and efforts for the people in AlgoSyn.

It was not only the exciting research that made the past five years a great time at RWTH Aachen but also my colleagues of which many had an impact on my research and work: Dr. Ibrahim Armac, Marita Breuer, Angelika Fleck, Dr. Hans Grönniger, Timo Greifenberg, Dr. Tim Gülke, Sylvia Gunder, Arne Haber, Dr. Thomas Heer, Lars Hermerschmidt, Dr. Christoph Herrmann, Katrin Hölldobler, Andreas Horst, Steffi Kaiser, Dennis Kirch, Dr. Anne-Thérèse Körtgen, Carsten Kolassa, Thomas Kurpick, Achim Lindt, Markus Look, Ulrich Loup, Dr. Shahar Maoz, Dr. Cem Mengi, Klaus Müller, Antonio Navarro Pérez, Pedram Mir Seyed Nazari, Johanna Nellen, Dr. Claas Pinkernell[1], Dimitri Plotnikov, Deni Raco, Holger Rendel, Dirk Reiss, Alexander Roth, Dr. Martin Schindler, Christoph Schulze, Galina Volkova, Michael von Wenckstern, Dr. Erhard Weinell, Dr. Elias Weingärtner, Dr. Ingo Weisemöller, Andreas Wortmann[1], and Ralf Zimmermann.

Finally, I am very grateful to my family for their support and the appreciation they show for my work.

[1]The best office mate.

Contents

Chapter 1.

Introduction

Interactive systems are systems that typically consist of multiple, to some extent independent, subsystems that collaboratively perform complex tasks [BS01]. The subsystems of interactive systems may include and depend on hardware and software. Examples of these systems are control units of aircrafts and automobiles, the software modules of cloud computing systems, and the hardware and software of robotic systems.

Separate components of interactive systems cooperate by exchanging messages. A component interacts with its environment by consuming and providing information. The environment of a component can again consist of multiple components or physical systems [Lee08].

The development of distributed interactive systems is an inherently complex and challenging task. Systems and subsystems typically operate in previously unknown cyber-physical environments, different configurations, and various compositions of subsystems. We are specifically interested in the software that controls these interactive systems and its logical architecture.

The software architecture of a system is the set of its *principal design decisions decisions* [TMD09]. A common approach to describe the logical decomposition of the software architecture of an interactive system are component and connector models [BS01, TMD09, CBB+10].

Component and connector models describe the organization of systems as a set of components interacting via connectors. Components encapsulate a system's functionality and offer it via explicitly defined ports and interfaces. Connectors enable and control the communication between components.

We are interested in the development and integration of suitable specification, analysis, and synthesis mechanisms for the structure and behavior of interactive component and connector systems.

Section 1.1 introduces the context of this thesis including a brief overview of C&C systems and model-based software development. We summarize related work for modeling structure and behavior of interactive C&C systems in Section 1.2. We state the objective of our research and highlight its main contributions to model-based development of interactive C&C systems in Section 1.3. Section 1.4 gives an overview of the structure of this thesis.

1.1. Context

Interactive systems [BDD+92, BS01] consist of multiple interacting components. These systems are also called distributed, embedded, or cyber-physical systems [Lee08, LS11]. Cyber-physical systems comprise software systems that interact with physical processes. We focus on the software part of these systems.

The software part of cyber-physical systems is again an interactive system. Its structure and organization is one of the concerns of software architecture [TMD09]. A common way of modeling the structure of interactive systems are component and connector (C&C) models [TMD09, CBB+10]. C&C models consist of components that encapsulate dedicated functionality. Every component has an interface that publishes information on the input required by the component and the output that it provides. Connectors connect the interfaces of components and enable component interaction. Components and connectors can be hierarchically composed to new components.

C&C models of interactive systems are often described using C&C architecture description languages (ADL) [TMD09]. Many integrated development tools, e.g., Matlab Simulink [wwwn], Ptolemy II [wwwaa], or AutoFOCUS [wwwe], provide related formalizations of components and connectors. The description of an interactive system requires not only a definition of its structure and decomposition into components but also a description of component behavior. These descriptions are provided as implementations in programming languages or as behavioral models.

We focus on both the structural and behavioral aspects of the software architecture of interactive C&C systems. A general model of the behavior of software components are discrete event systems [BS01, CL07]. The interaction of components can be formalized as observations of the messages received and the messages sent by components. These communication histories are formalized as traces or message streams [BS01, BK08, LS11, RR11]. Convenient mechanisms for describing component behavior are state-based modeling languages that describe a component's interaction with its environment. Examples for these description techniques are I/O automata [LT89], Statecharts [Har87], and Stateflow diagrams [wwwo].

Model-based software development centers around using models throughout the software development process [Sch06, SVB+06, Rum12]. Models are constructs that refer to selected aspects and attributes of originals [Sta73], e.g., a model of a component refers to its manifestation in the software system. A model might selectively focus on the component's interface, its behavior, or both. In model-based development, models are employed throughout the development process for requirements specification, testing, model checking, and code generation. We are interested in model-based analyses where a specification and a system are defined by sets of models and an analysis determines whether the system satisfies the specification. In the case of synthesis, a specification is provided and an algorithm computes a satisfying system, if one exists.

Our work on analysis and synthesis methods for interactive C&C systems builds on the ADL MontiArc [HRR12], the Focus [BS01] stream processing framework, and the MontiCore framework [KRV10, wwwv]:

- We formalize C&C models using the ADL MontiArc [HRR12, wwwq]. Language features of the ADL MontiArc to describe software architecture models include hierarchical composition of components, component type definitions, instance declarations for component reuse, and message type definitions using UML/P class diagrams [Rum11, Sch12].

- We use FOCUS as a foundational theory and semantic domain for interactive C&C systems. FOCUS provides a system development methodology based on behavior specifications and stepwise refinement of interactive systems. For the definition of component behavior, we employ time-synchronous streams and state-based models following the I/O^{ω} automata paradigm [Rum96] with stream processing semantics. Our notion of refinement is based on component refinement as defined in [Bro93] and [BS01].

- The modeling and specification languages presented throughout this thesis are implemented using the MontiCore framework for the development of textual domain-specific languages and tools. Our work extends and employs existing MontiCore languages, e.g., MontiArc and UML/P. The analysis, synthesis, and code generation prototypes implemented make use of the MontiCore symbol table framework [Vö11] and the MontiCore code generation framework [Sch12].

The field of cyber-physical systems deals with many heterogeneous concerns and so does software architecture. Software architecture concerns not addressed in our work are descriptions of continuous processes and topics such as realtime properties, timing constraints, and memory or other resource consumption. These issues form important research fields of their own [TMD09, Lee09, ABG+13].

1.2. Related Work on Component and Connector Modeling and Analyses

We briefly review related work in the area of modeling and specification languages for component and connector systems. The first part of this section provides an overview of description mechanisms for the structure of C&C systems and related analysis and synthesis methods. The second part deals with the definition of component behavior for the description and analyses of interactive systems.

1.2.1. Component and Connector Structure Descriptions

A common approach to describing the structure of distributed interactive systems and their hierarchical decomposition are component and connector models [TMD09, CBB+10]. The organization of a system into components that encapsulate functionality and connectors that enable interaction are common in architecture description languages, general system modeling languages and tools. In the following we describe some ADLs from the

literature and describe their modeling capabilities for C&C models. A deeper discussion of these languages in the context of our work can be found in Section 2.4.

The Architecture Analysis Design Language (AADL) [wwwa, FGH06, FG12] is an architecture description language standardized by the Society for Automotive Engineers. AADL models contain component type declarations and implementation declarations. A special design concept of AADL are predefined component categories divided into application software, execution platform, and generic components. This domain-specific orientation of AADL makes it suitable for embedded realtime systems especially in the automotive, avionics, and aerospace domains [wwwa, HWF+10, BCK+11, FG12].

Acme [GMW00, SG04] is an ADL that was initially developed as an architecture language interchange format [GMW97] containing the common elements of various ADLs. These elements comprise components, ports and connectors. Acme has evolved to a stand alone ADL. One of the main features of Acme and its tool support AcmeStudio [wwwy] is the definition and analysis of patterns of architectural elements called *architectural styles* [SG04, KG10].

ArchiMate [LPJ10, JPL+11] is a modeling language for enterprise architecture. In a recent survey on the use of ADLs in industry [MLM+13] it was reported the second most used ADL after UML [Obj12a]. ArchiMate divides the description of a system into three layers: the business layer, the application layer, and the technology layer. The application layer contains models of the application software that are organized in components. Components can interact via provided and required application interfaces or via application collaboration [The12].

MontiArc [HRR12] is an ADL designed for modeling cyber-physical systems [Lee08]. The ADL is developed based on the core ADL concepts identified in [MT00] and its semantics is based on the stream processing theory FOCUS [BS01]. MontiArc is a textual ADL developed using the language workbench MontiCore [KRV10]. One important feature of MontiArc is its typing and instantiation mechanism for components. Haber et al. [HRRS11, HRR+11, HRRS12] have extended MontiArc with support for modeling variability in the context of software product lines.

Many works also consider the Unified Modeling Language (UML) [Obj12a] an architecture description language [ICG+04, MDT07, MLM+13]. The UML consists of 14 diagram types for modeling the structure and the behavior of software systems. Ivers et al. [ICG+04] describe how to use UML 1.4 and UML 2.0 for documenting component and connector models.

The Systems Modeling Language (SysML) [FMS11, Obj12b, Wei07] is a derivation of UML 2 defined as a profile using UML's profile mechanism [Obj12a]. SysML introduces block definition diagrams (based on UML class diagrams) and internal block diagrams (based on UML composite structure diagrams). It extends ports and flows of UML and thereby overcomes some of the criticized weaknesses of UML for the definition of component and connector models.

A variety of tools implements C&C models. One popular example is the block diagram language implemented in MathWorks Simulink [wwwn]. Block diagrams consist of blocks and lines. In the terminology of C&C models blocks are components, signals and ports

correspond to ports, and lines correspond to connectors. Another implementation of C&C models is provided within the AutoFOCUS tool suite [BHS99, HF07, wwwe]. C&C models are modeled as component architectures consisting of components with typed input and output ports that are connected via channels (connectors).

Our work is based on these popular description mechanism and their separation of systems into components and connectors. We intentionally focus on the pure and clean concepts of C&C models common to these description techniques.

1.2.2. Structural Specification Mechanisms and Analyses

For many use cases a complete hierarchical implementation-oriented description of a system and its subsystems as a C&C model does not provide the right level of abstraction to present relevant details and context information. In early design phases some details of components and connectors might still be unknown. For documenting important design decisions the description of some components and ports may be irrelevant or distracting. We now reflect on existing specification mechanisms for C&C models, related analyses, and their support for abstraction and refinement. A deeper discussion of these topics in the context of our work can be found in Section 3.7, Section 4.6, and Section 5.8.

AADL [FGH06, FG12] supports specifications with incomplete information of port types and with abstract flows, which show the source and sink of flows but not their complete path through the system. We have not found previous works on checking the structure of AADL architectures against specifications including abstractions of connectors and of component hierarchies.

Armani [Mon98, Mon99] is a framework to define architectural styles and design rules for architectures. Armani is based on the ADL Acme [GMW00, GMW97]. An engineer can define styles and rules for systems using a constraint language based on first order predicate logic. Example predicates include $connected(c_1, c_2)$ and $reachable(c_1, c_2)$, which assert connectedness and transitive connectedness of components c_1 and c_2. Armani's constraints are evaluated over concrete architectures. The constraint language is integrated into AcmeStudio [wwwy] and constraints are automatically evaluated while editing architectures. To the best of our knowledge, the language neither supports a transitive subcomponent relation nor supports constructs to crosscut the bounds of the traditional implementation-based hierarchical decomposition of systems to their subsystems, types to subtypes etc.

Bhave et al. [BKGS11] have extended AcmeStudio to support structural consistency between heterogeneous models as architectural views, specifically for cyber-physical systems. View consistency is checked by verifying if a morphism exists between two typed graphs.

Chechik et al. [FBDCS11, SFC12] introduce a language independent mechanism for incomplete models. Their approach operates on the syntax definition of a modeling language and adds annotations to mark an element as optional (*May*), as abstract representing sets of concrete elements (*Abs*), as possibly identical with another element (*Var*), and to mark the whole model as incomplete (*OW*). The approach is generic and language independent. It focuses on syntactic incompleteness of models.

Many works in software architecture deal with refinement relations between architectural elements. Broy and Stølen [Bro93, BS01] introduce glass-box refinement where a refinement has to respect the decomposition of a specification into components and directed channels between them. Philipps and Rumpe [PR97, PR99] have developed a set of proven refinement rules based on [Bro93].

Previous work in our group [GHK+08b, GHK+08a] described the use of views with a focus on the automotive domain, using SysML's internal block diagrams. SysML's internal block diagrams provide under-specification mechanisms for component hierarchy and connectivity. However, the question of verifying the structure of a C&C model against a view is discussed neither in these works nor in any other SysML related work we have found.

Boucké et al. [BWH10] present composition operators for C&C models to avoid the repetition of elements in integrated models. The integration of sets of architecture models is based on a user specified unification relation (marking identical elements across input models) and a submodel relation (e.g., a component is detailed in another model). This model composition is an imperative composition of concrete C&C models.

Giese and Vilbig [GV06] discuss separation of non-orthogonal concerns in software architecture and design. Architectural views are defined as directed graphs representing components and connectors extended with behavior contracts. The structural part of architectural view composition can be handled by superposition of these directed graphs [GV06]. The work does not allow abstraction of direct hierarchy and connectivity.

Sabetzadeh and Easterbrook introduce a graph based framework for view merging for arbitrary modeling languages [SE04, SE06]. Their framework is modeling language independent. The merging of multiple views integrates the information contained in several partial views. One weakness of the framework is that it does not handle complex well-formedness rules of the views merged. A merge of views using the framework from [SE06] might result in a structure that is neither a view nor a C&C model.

Common limitations of structural specification and analyses mechanisms

Generic, syntax-based approaches [SE06, BKGS11, FBDCS11] can be applied to a wide range of modeling and description languages but do not allow domain specific abstractions with semantics beyond generic syntactic criteria. The Armani [Mon99] specification language integrated with Acme allows expressing well-formedness rules and predicates in the implementation-oriented view of ADLs, i.e., not crosscutting the component hierarchy or multiple components. To the best of our knowledge the AADL [FG12] is the only ADL that directly supports refinement statements for architectural elements. Checking refinements seems however not to be automated.

In general, the existing works do not provide C&C model specific means for documenting partial knowledge and design decisions while selectively abstracting away implementation-specific information such as direct component hierarchies, component interfaces, and connectivity. Currently, integrating the knowledge and designs of multiple system descriptions requires user interaction [BWH10], might not lead to well-formed

models [SE06], or restricts specification mechanisms available to the modeler [GV06, FBDCS11].

1.2.3. Interactive Systems Behavior Descriptions

We now briefly review description mechanisms for modeling the behavior of interactive systems. Some of these constitute basic models of computation with unambiguous semantics while others are modeling languages with no or incomplete formal semantics definitions. We give an overview of these languages. A deeper discussion in the context of our work can be found in Section 6.6.

I/O automata by Lynch and Tuttle [LT89] model components that are executed in parallel and communicate by executing input and output actions. Each transition is labeled with one action. Transitions labeled with input actions can be executed when the identical action is executed as an output action by another automaton. The composition of multiple I/O automata requires the disjointness of the sets of output actions of different automata because the execution of each action may only be controlled by one component.

Alfaro and Henzinger [dAH01] introduced interface automata to describe possible compositions of components. Interface automata capture input assumptions and output guarantees of components. A set of components can be composed if there is at least one environment that fulfills the assumptions of the composition. The structure of interface automata is similar to I/O automata.

Constraint automata [BSAR06] are an extension of I/O automata. The transitions of constraint automata are guarded with sets of active channels and constraints over the data on these channels. Another extension are message-passing automata [BL01, BL06] that describes communicating components. A message passing automaton consists of a finite set of local automata that represent components. Components are connected pairwise by reliable channels of unbounded size that allow an asynchronous FIFO communication [BL06].

Statecharts by Harel [HP85, Har87] are one of the most prominent visual description techniques for reactive systems. The language provides many features, e.g., hierarchical states, action execution in states, receiving and sending signals. Statecharts are standardized both in UML [Obj12a] and SysML [Obj12b]. As described before these languages also offer the modeling of C&C models. SysML allows for defining bindings and allocations to relate statecharts to components in block diagrams and thus enables behavior modeling for C&C models.

The AutoFOCUS tool [HSSS96, HF07] for the specification and prototyping of distributed systems allows behavior definition of components using state transition diagrams. These can be edited in a graphical representation and translated into executable simulation code. Transitions read messages on input ports of components and send messages on output ports.

The block diagram language implemented in MathWorks Simulink [wwwn] is extended with state transition diagrams in Stateflow [wwwo]. The automata of Stateflow are a combination of Mealy and Moore machines and they are fully integrated in the simulation and code generation environment of Stateflow. The semantics of Stateflow diagrams is

only given informally but has been formalized in many ways by various translations into other formalisms [MC12].

For the modeling of interactive C&C systems we are mainly interested in modeling languages that are syntactically and semantically integrated with component models such as Stateflow and AutoFOCUS' transition diagrams. These languages provide domain specific concepts, e.g., receiving and sending messages via ports of components.

1.2.4. Underspecification Mechanisms and Refinement

An important concept of behavioral models is underspecification, which allows to leave open some details in a specification. Underspecification enables iterative development through step-wise refinement [BS01]. Automated refinement checking assists in developing an implementation that conforms to, i.e., refines, its specification. We review specification mechanism of state-based behavior models below. A deeper discussion in the context of our work can be found in Section 6.6 and Section 7.8.

Underspecification can be modeled by distinguishing possible from required behavior. This distinction is an integral part of modal transition systems (MTS) [LT88, Lar89] where transitions are marked as *may* or *must*. Informally, a refining MTS has to preserve all *must* transitions of the MTS it refines. The refining MTS can preserve *may* transitions, change them to *must* transitions, or remove them. The classic refinement relations for MTS is strong modal refinement [LT88]: the corresponding states of the refining MTS can simulate all *must* transitions and the states of the more abstract MTS can simulate all *may* transitions. Further refinement relations are defined for MTS [LNW07b].

Interface automata [dAH01], which distinguish between input and output, define a refinement based on alternating simulation [AHKV98]. The more concrete automaton can simulate all input steps of the abstract one while the abstract can simulate all output steps of the concrete, i.e., an implementation may allow more inputs and less outputs.

Differently, Baier et al. [BSAR06] relate constraint automata to timed data stream semantics and define refinement of constraint automata as the containment of relations over timed data streams in the semantics of the constraint automata. Thus, the accepted input may not be extended as part of a refinement.

This notion of refinement is also defined as behavioral refinement in the FOCUS framework [Bro93, BS01]. Scholz [Sch98] has defined a semantics based on FOCUS' message streams for a subset of Harel's statecharts with a refinement based on inclusion of the relations of input and output streams. A different refinement of statecharts for modeling object oriented systems has been defined by Harel and Kupferman [HK02]. The refinement is based on the object oriented `is-a` relation and thus requires that a refined statechart has every behavior the abstract statechart has. The refinement thus may only add behavior while FOCUS considers a refinement that removes behavior (remove uncertainty).

Common limitations

Many formalisms exist for the state-based modeling of behavior. Some of these have no explicit composition mechanisms, define composition based on common actions, or composition based on recognized input and output messages. Refinement is a well-understood concept for many formalisms but lacks thorough integration into higher-level modeling languages used in combination with C&C models such as UML and SysML statecharts, AutoFOCUS' diagrams, and Stateflow. In some cases a lack of well-defined semantics and underspecification mechanisms hampers possible automation and assistance of engineers.

To the best of our knowledge no existing solution provides high-level modeling languages with rich syntactic features for integrated C&C structure and behavior modeling with automated refinement checking for iterative system development.

1.3. Objective and Main Results

Current approaches to model-based development of interactive C&C systems follow the traditional, implementation-oriented hierarchical decomposition of systems into subsystems and components. This approach limits the possibilities of stakeholders and engineers to document and express concerns in C&C models that crosscut the boundaries of components or subsystems.

We investigate modeling languages that allow to express partial and crosscutting knowledge about the structure and behavior of interactive C&C systems. For these modeling languages, we identify and formally define analysis and synthesis problems resulting from the newly available specification mechanisms.

The research objective of this thesis is to support the model-based development and evolution of distributed interactive systems. The goal is to provide domain-specific notations with corresponding analysis and synthesis methods to guarantee the correctness of implementations.

We introduce modeling languages, methods, and tools applicable in different stages of system development. The introduced modeling languages and language profiles allow to express requirements and partial knowledge available about the structure and behavior of interactive C&C systems. We define structural specifications, analysis, and synthesis methods for C&C models. Behavior of components in C&C software architectures is modeled using MontiArcAutomaton automata following the I/O^{ω} automata paradigm [Rum96]. Based on these behavior specifications, we have developed methods for automated verification and refinement checking. In addition to verification, the MontiArcAutomaton framework also supports code generation, e.g., for robotic systems.

The contributions presented in this thesis are:

- We have developed a language profile of the MontiArc modeling language for C&C views, which includes novel abstraction mechanisms to describe the structure of C&C models. C&C views crosscut the traditional boundaries of the implementation-oriented hierarchical decomposition of systems and subsystems by providing ab-

stractions over direct hierarchy, direct connectivity, port names and types. The concrete syntax for C&C views is similar to that of the ADL MontiArc and thus allows a 'by example' description of C&C models.

- Given a candidate or evolved C&C model of a software architecture and a set of C&C views, an engineer can check whether the architecture satisfies the views. In addition to a Boolean verification result, our algorithms generate informative witnesses, for inspection by the engineer, that demonstrate the reasons for satisfaction or non-satisfaction. We have presented the verification of C&C models against C&C views in [MRR14].

- Given a set of C&C views and a propositional formula over these views describing required, alternative, dependent, and forbidden designs, an engineer can automatically synthesize a satisfying architecture, if one exists. Our synthesis method is based on a reduction to a Boolean satisfiability problem via Alloy [Jac06]. We have presented the synthesis of a C&C model from C&C views in [MRR13].

- We have developed a modeling language for a state-based description of the input and output behavior of components. The description mechanism allows underspecification and capturing partial knowledge available about a component's behavior. This language is introduced in [RRW12] and [RRW14].

- Given behavior descriptions of components, an engineer can automatically check whether a modeled system satisfies its specification. The implemented analysis methods also allow to check whether one subsystem can replace an other subsystem in a given context.

- One result of the analysis framework for checking component equivalence and refinement is a generic translation of component behavior to the model checker Mona [EKM98] that allows the definition of further analyses.

- Given a complete description of a system's C&C architecture and the behavior of its atomic components, our code generator generates Java code for deployment on the leJOS Lego NXT robotics platform. We have previously reported on our code generation framework in [RRW13b].

- We have implemented and evaluated the analysis and synthesis methods for C&C models on various example systems. We have assessed the helpfulness of generated witnesses for C&C views verification in a user study. In addition we have evaluated the modeling and code generation framework for programming distributed, autonomous robotic systems in a one-semester workshop class with eight master students. We have reported observations from this case study in [RRW13a].

1.4. Thesis Organization

We introduce the basic concepts of C&C software architectures, define C&C models, and introduce the modeling language MontiArc in Chapter 2. We discuss similar ADLs and related modeling languages.

In Chapter 3, we introduce C&C views and a profile of the modeling language MontiArc to document these views. The semantics of a C&C view is defined by the C&C models satisfying the C&C view.

We present algorithms to check whether a C&C model satisfies a C&C view in Chapter 4. We identify reasons for non-satisfaction and present techniques to compute witnesses that demonstrate positive and negative verification results.

Chapter 5 investigates the problem of synthesizing a C&C model from a C&C views specification and presents our solution based on a translation into Alloy. We extend our solution for supporting advanced features such as library components and architectural styles.

Chapter 6 presents the modeling language MontiArcAutomaton for modeling behavior of components as state machines. We introduce a language profile for synchronous communication and its semantics based on the FOCUS theory of streams.

We present a translation of C&C models and time-synchronous automata into the decidable logic WS1S [Tho90] in Chapter 7. Our prototype implementation uses Mona, which implements a decision procedure for WS1S, for checking equality and refinement of systems and subsystems.

In Chapter 8, we present a framework and code generator for generating executable Java code from MontiArcAutomaton models. As one example application, the generated code is deployed to Lego NXT robotic systems.

Evaluation based on example systems and a case study are presented within every chapter that introduces novel techniques and prototype implementations.

Chapter 9 concludes the thesis and outlines open research questions for future work.

Chapter 2.

Component and Connector Software Architectures

Software architecture deals with high-level descriptions of a software system structure and behavior. Taylor et al. [TMD09] give a broad definition of the term software architecture as *the set of principal design decisions about the system*. Following their definition, the architecture of a software systems contains, e.g., decisions about the structure, behavior, interaction, and non-functional properties of the system. We are most interested in the structure and behavior of software systems and ways to formalize both to support engineers and architects in developing software systems.

Specifically, we are interested in component and connector (C&C) architectures, which are used in many application domains, from cyber-physical and embedded systems to web services to enterprise applications [BDD+92, BS01, Bro05, BR07, TMD09]. C&C architectures offer a physically distributed computation model as well as a logically distributed development process of components with well-defined interfaces. A C&C model consists of components at different containment levels, their typed input and output ports, and the connectors between them.

Components are C&C model elements that provide and encapsulate services. Provided services are, e.g., the computation or storage of data. Components provide and require services via publicly published interfaces. An example for a component is, e.g., an arbiter that requires input from two sources and provides a single output based on the two inputs. Figure 2.1 shows an arbiter component with the component name `ModeArbiter`. The arbiter is contained in its parent component `Controller`.

Connectors are C&C model elements that enable the interaction of components. In the example of the `ModeArbiter`, two incoming connectors connect the two sources required by the arbiter with its input ports. In logical software architectures we treat components and connectors as abstract elements independent of concretizations as pieces of hardware, buses, or wires connecting components.

An example for a C&C model consisting of four components is shown in Figure 2.1. The component named `Controller` has two incoming ports and a single outgoing port. It contains three subcomponents: component `UserOperation`, component `EMS-Operation`, and component `ModeArbiter`. The three subcomponents may be atomic components or further decomposed. The example in Figure 2.1 shows an abstraction over the implementation details of components inherent to C&C models. For the composition of components only the (type) names and interfaces of the subcomponents are relevant

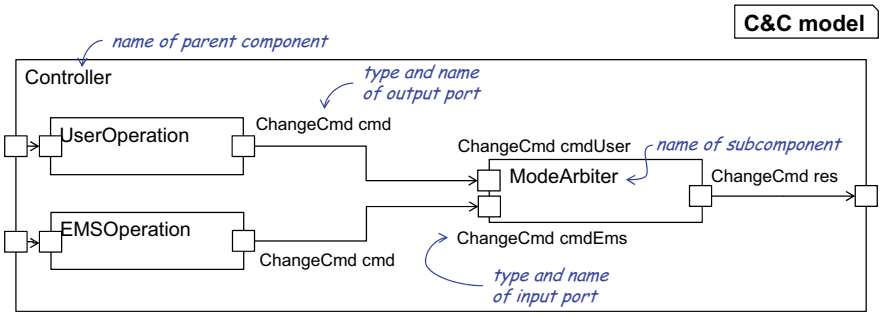


Figure 2.1.: A simple C&C model — consisting of a composed component with three subcomponents — shown in the graphical syntax of the ADL MontiArc.

and not their implementations.

The components `UserOperation` and `EMSOperation` both have an outgoing port of the type `ChangeCmd` named `cmd` which is connected to an incoming port of the component `ModeArbiter`. The names of ports are unique in the context of their component. A connector forwards all messages from the output port `res` of the component `ModeArbiter` to the single output port of the component `Controller`. Please note that all ports have names but we omit them inside Figure 2.1 to avoid clutter. The only communication allowed by the configuration shown in Figure 2.1 is communication via the directed connection between the component's ports.

Chapter outline and contributions

Our main contribution presented in this chapter is the formal definition of C&C models in Section 2.1 and a language profile of the ADL MontiArc to provide a concrete syntax for C&C models illustrated in Section 2.2. Section 2.3 explains the difference between C&C models and component type definitions in MontiArc. We conclude this chapter with an overview of C&C modeling in other ADLs and modeling notations in Section 2.4.

2.1. Component and Connector Models

Based on component and connector models as described in [MT00], as used in the Focus framework [BS01], and as formalized by the ADL MontiArc [HRR12], we now define the basic structure of C&C models used in this thesis.

Component and connector models consist of components at different containment levels, their directed, typed, and named ports as well as connectors that connect these ports. We formally define C&C models in Definition 2.2. The first part of Definition 2.2 describes the main elements of C&C models: components, ports, connectors, and port types. The second part defines well-formedness rules of C&C models as requirements on relations between C&C model elements.

Definition 2.2 (Component and connector model)**.** A Component and Connector model m is a structure $m = (Cmps, Ports, PNames, Types, Cons, subs, ports, dir, type, name)$ where

1. $Cmps$ is a set of components $cmp \in Cmps$ (with unique names), each of which has a set of ports $ports(cmp) \subseteq Ports$ and a (possibly empty) set of immediate subcomponents $subs(cmp) \subset Cmps$,

2. $Ports$ is a set of directed input and output ports $p \in Ports$ with $dir(p) \in \{IN, OUT\}$ where each port has a name $name(p) \in PNames$, a type $type(p) \in Types$, and belongs to exactly one component $\exists! cmp \in Cmps : p \in ports(cmp)$,

3. $Cons$ is a set of directed connectors $con \in Cons$, each of which connects two ports $(con.srcPort \in ports(con.srcCmp)$ and $con.tgtPort \in ports(con.tgtCmp))$ of the same type $type(con.srcPort) = type(con.tgtPort)$, and

4. $Types$ is a finite set of types $t \in Types$ that appear on ports: $t \in Types \Leftrightarrow \exists p \in Ports : type(p) = t$.

Additionally, the following rules for well-formedness apply:

5. $\nexists c \in Cmps : c \in subs^+(c)$, where $subs^+$ denotes the transitive closure of the subcomponent relation $subs : Cmps \times Cmps$, i.e., no component is its own (transitive) parent,

6. $\forall child \in Cmps : |\{parent \in Cmps \mid child \in subs(parent)\}| \leq 1$, i.e., every component has at most one direct parent,

7. $\forall cmp \in Cmps : \forall p_1, p_2 \in ports(cmp) : name(p_1) = name(p_2) \Rightarrow p_1 = p_2$, i.e., port names are unique within each component, and

8. $\forall p \in Ports : |\{con \in Cons \mid con.tgtPort = p\}| \leq 1$, i.e., every port has at most one incoming connector, and

9. $\forall con \in Cons$ we have exactly one of the four cases
   (a) $con.srcCmp = con.tgtCmp \wedge dir(con.srcPort) \neq dir(con.tgtPort)$, i.e., a component forwards input directly as output or feeds back its own output as input,
   (b) $\exists parent \in Cmps : \{con.srcCmp, con.tgtCmp\} \subseteq subs(parent) \wedge con.srcCmp \neq con.tgtCmp \wedge dir(con.srcPort) = OUT \wedge dir(con.tgtPort) = IN$, i.e., two sibling components with a common parent are connected,
   (c) $con.tgtCmp \in subs(con.srcCmp) \wedge dir(con.srcPort) = IN \wedge dir(con.tgtPort) = IN$, i.e., a component forwards input to an immediate child,
   (d) $con.srcCmp \in subs(con.tgtCmp) \wedge dir(con.srcPort) = OUT \wedge dir(con.tgtPort) = OUT$, i.e., a component forwards output from an immediate child.

△

Notation: For a C&C model m, a port $p \in m.Ports$, and a component $c \in m.Cmps$ we use the short notation $p \in c.ports$ to denote $p \in m.ports(c)$ in case the C&C model m is clear from the context. For a component $c \in Cmps$ we write $c.Ports_{IN}$ to refer to its input ports $\{p \in c.Ports \mid p.dir = IN\}$ (respectively $c.Ports_{OUT}$ for its output ports). In addition, we write $m.name$ and $c.name$ (for $c \in Cmps$) for the unique names of C&C models and components.

Without loss of generality, we consider only C&C models with exactly one top component, i.e., $\exists! cmp \in Cmps : \nexists parent \in Cmps : cmp \in subs(parent)$. The second part of Definition 2.2 lists important rules for the well-formedness of C&C models. Together with the requirement of a single top component the subcomponent relation *subs* forms a tree since it has no cycles (see Definition 2.2, Item 5).

In Definition 2.2, Item 4 we define the set of types for a C&C model m, i.e., $m.Types$, as the types that appear on ports. An alternative to this definition would allow the set $m.Types$ to contain types not used in the C&C model, e.g., a set of common basic types. We apply this restriction without loss of generality since it simplifies the definition of the set $m.Types$ in the concrete syntax of C&C models and the completeness proof of our verification algorithm in Section 4.3.5.

The four cases of valid connectors in C&C models are listed under Item 9 of Definition 2.2. No connector may cross the boundaries of the parent component or the subcomponents. As a concrete example consider Figure 2.3: a connector from the output port of component `ModeArbiter` connecting to the input port of the component `Actuator` is not allowed by Definition 2.2. In the C&C model from Figure 2.3 the connection is realized by two connectors: one connector from the output port of the component `ModeArbiter` to the output port of its direct parent component `Controller` (connector of type Definition 2.2, Item 9 (d) and a second connector from the output port of the component `Controller` to the input port of the component `Actuator` (connector of type Definition 2.2, Item 9 (b).

Please note that the structure of C&C models given in Definition 2.2 describes concrete component and connector models, i.e., the instances of components and their composition. Some ADLs, e.g., the AADL [FG12], Acme [GMW00], and MontiArc [HRR12], separate the definition of components from the description of their instantiation and composition to systems (C&C models). This separation is supported, e.g., by instantiation mechanisms used in AADL and MontiArc or the explicit models for *systems* and *representations* used in Acme. These mechanisms simplify the reuse of components over multiple C&C models and also allow the reuse in the same C&C model. We briefly discuss the relation of component definitions and their instantiation to C&C models in the case of MontiArc in Section 2.3.

We use a subset of the ADL MontiArc as a concrete syntax to model the C&C models from Definition 2.2 as demonstrated in Section 2.2.

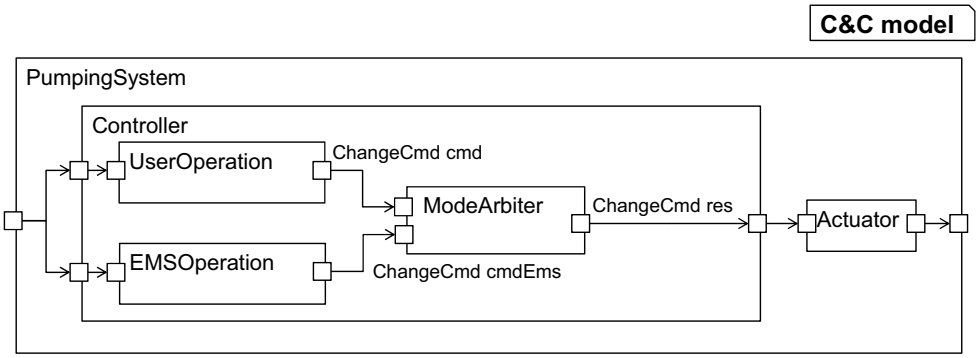


Figure 2.3.: The C&C model `PumpingSystem` shown in graphical syntax. Some details, e.g., the types and names of ports are omitted to avoid clutter. An excerpt of the textual MontiArc syntax of this example is shown in Listing 2.4.

2.2. Modeling Component and Connector Models Using MontiArc

We now give a brief overview of the concrete textual syntax of MontiArc for modeling C&C models. The following description is based on the MontiArc language reference [HRR12] with additional examples of MontiArc's concrete syntax.

The basic elements of MontiArc are components with ports, subcomponents, and connectors that unidirectionally connect ports of components. The interface of a component is a set of typed and directed ports. The internal structure of components can be defined by the decomposition of a component to subcomponents and connectors.

For the description of C&C models we ignore the following MontiArc language features: component types, referencing and instantiation, generic component types, and configurable components. These features are described in the MontiArc technical report [HRR12] with examples for the concrete textual syntax. We discuss the instantiation mechanism and component type definitions as used in MontiArc in Section 2.3 and Section 6.2.1.

2.2.1. Components and Subcomponents

An example of a C&C model in concrete textual MontiArc syntax is shown in Listing 2.4. The listing shows excerpts of the C&C model `PumpingSystem` also shown in Figure 2.3. Every MontiArc artifact (file that contains the model) starts with a `package` declaration (see Listing 2.4, l. 1). The outer most element of MontiArc models is a component definition (line 3). The package name `pumpStationExample` and the component name `PumpingSystem` are the unique coordinates of the MontiArc model.

Listing 2.4 shows the C&C model `PumpingSystem` named after its top component

PumpingSystem. The C&C model contains the components Controller (line 8) and Actuator (line 34) as direct subcomponents of component PumpingSystem. The subcomponent Controller consists of the components UserOperation, EMSOperation, and ModeArbiter. The subcomponent Actuator of the component PumpingSystem is not further decomposed.

Components are modeled using the keyword component. Syntactically, there is no difference between subcomponents and parent components except that subcomponents are defined inside the body of their parent component (see, e.g., line 11, line 17, and line 23 of Listing 2.4 for three subcomponents defined inside the body of the component Controller). Since the subcomponent relation of C&C models has no cycles (see Definition 2.2, Item 5) we exclusively and completely express it by the inclusion of the subcomponents in the textual MontiArc C&C models.

2.2.2. Interfaces and Ports

The interface of a component is the set of its incoming and outgoing typed ports.

The interface of the component PumpingSystem (lines 4-6) consists of an incoming port with the name ready of the type Boolean and an outgoing port with name act, which also sends messages of the type Boolean. The ports of each component are defined inside the body of the component, e.g., the ports of the component PumpingSystem are defined in lines 4-6 and the ports of the component ModeArbiter are defined in lines 24-27. The definition of a single port or a set of ports starts with the keyword port the direction of the ports is given by the keywords in and out followed by the port's type and name. Multiple ports of one component can each be defined in a separate port statement (e.g., lines 12-13) or in one port statement separated by commas (e.g., lines 24-27).

The definition of ports has to respect the well-formedness rules in Definition 2.2, e.g., the names of all ports of one component have to be unique as defined in Definition 2.2, Item 7.

For a C&C model m we define the set $m.Types$ as the set of all type names that appear on ports. This ensures the constraint that all types appear on ports (Definition 2.2, Item 4) and it makes the explicit definition of the set $m.Types$ of all types unnecessary.

In addition it is of course possible to define all valid port types in an accompanying class diagram and use it to check the existence and compatibility of port types. This feature is supported by our implementation of behavior analysis in Chapter 7 and the code generation presented in Chapter 8.

2.2.3. Connectors

Connectors in C&C models connect two ports of a single component, two sibling components, or a parent and child component. Connected ports are the only means for interaction in C&C models. For the definition of C&C models, connectors are always placed inside their parent component and reference the ports of the parent unqualified while qualifying subcomponent's ports with their component names.

MontiArc

```
package pumpStationExample;

component PumpingSystem {
  port
    in Boolean ready,
    out Boolean act;

  component Controller {
    //...

    component UserOperation {
      port
        out ChangeCmd cmd;
        //...
    }

    component EMSOperation {
      port
        out ChangeCmd cmd;
        //...
    }

    component ModeArbiter {
      port
        in ChangeCmd cmdUser,
        in ChangeCmd cmdEms,
        out ChangeCmd res;
    }

    connect UserOperation.cmd -> ModeArbiter.userCmd;
    //...
  }

  component Actuator {
    port
      //...
      out Boolean pAct;
  }

  connect Actuator.pAct -> act;
  //...
}
```

Listing 2.4: An excerpt from the C&C model `PumpingSystem` given in MontiArc textual syntax. The C&C model is shown in graphical syntax in Figure 2.3.

The connector in line 30 of Listing 2.4 is placed inside the body of the composed component `Controller` and connects the port `cmd` of the subcomponent `UserOperation` with the port `userCmd` of its sibling component `ModeArbiter`. This connector is of the kind given in Definition 2.2, Item 9 (b).

The connector in line 40 of Listing 2.4 connects the port `pAct` of the subcomponent `Actuator` with the port `act` of the parent component `PumpingSystem`. This connector is of the kind given in Definition 2.2, Item 9 (d).

For the purpose of presentation and without loss of generality we only consider C&C models that are — with all subcomponents and connectors — completely defined within one file. Please note that MontiArc also supports component definitions in separate files down to one component per file. MontiArc offers suitable import and instantiation mechanisms that are essential for the reuse of models.

2.3. Component Type Definitions and Component and Connector Models

Component and connector models as introduced in Definition 2.2 consist of components and connectors with unique identities. For components solving recurring and generalizable problems, it is useful to extract component types that can be instantiated and reused. Many modeling languages in the architecture modeling domain have mechanisms that allow the definition of component types [TMD09, HRR12, FG12].

We focus on component type definitions and the instantiation mechanism as implemented in the ADL MontiArc [HRR12].

The component type definitions expressed in MontiArc appear very similar to C&C models. The main difference is that a component type definition defines not the identity of a concrete instance in a C&C model but a more general component type. A component type definition consists of the interface of a component and its possible decomposition. The decomposition modeled in a component type definition is restricted to one level of hierarchy (it is however technically possible to model more than one component type in a single artifact [HRR12]).

Subcomponent references in component type definitions are definitions of component instances consisting of a subcomponent name and a reference to the component type definition of the subcomponent. An example of the composed component type definition `TwoSwitchController` is shown in Figure 2.5 (a). The composed component type has two subcomponents of the component type `ToggleSensor` and one subcomponent of the component type `Controller`. The component type definition of the component type `ToggleSensor` is shown in Figure 2.5 (b).

A component type can be instantiated to a C&C model. When instantiating a component of the type `TwoSwitchController` from Figure 2.5 (a) a component instance is created that consists of the composition of the instances of all subcomponents of the component `TwoSwitchController`. The instance of the component type `TwoSwitch-Controller` is shown as the C&C model `twoSwitchController` in Figure 2.5 (c). Since C&C models as defined in Definition 2.2 do not share the concept of component

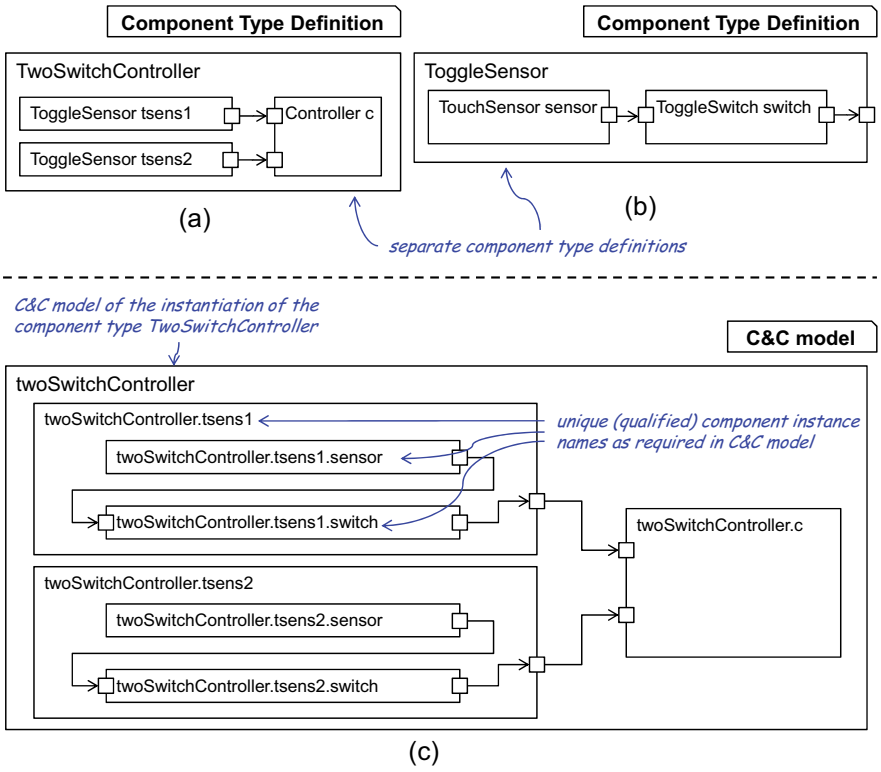


Figure 2.5.: An instantiation of component type definitions to C&C models.

types and all components in a C&C model require a unique name, we have chosen the names of the components (fully qualified) based on their subcomponent definition instance names from the component type definitions in this example. Different namings are possible.

We formally define the structure of component type definitions for MontiArcAutomaton components in Definition 6.8 in Section 6.2.1. The modeling language MontiArcAutomaton is based on MontiArc and extends component type definitions with variables.

For the first part of this thesis in Chapters 2-5 we focus on C&C models as presented in Definition 2.2. We extend our focus to component type definitions in Chapters 6-8.

2.4. Related Component and Connector Architecture Description Languages

C&C models are an integral part of many architecture description languages. Early classification frameworks for ADLs [GS93, KC94, Cle96, MT00] characterized ADLs by the

ability to represent components and connectors. Medvidovic and Taylor [MT00] define the main features of ADLs as components with interfaces, connectors, and architectural configurations. MontiArc and the subset of MontiArc elements we use to express C&C models is an ADL according to these definitions. A component is a unit of computation, an interface is the set of interaction points defined for a component, connectors model component interaction and the rules of the interaction. Architectural configurations are graphs that describe the composition of components and connectors [MT00]. In our terms interfaces are the set of ports of a component in C&C models, the architectural configuration is given by the connectors and the subcomponent relation in C&C models.

More recent characterizations of ADLs include more viewpoints than just the C&C viewpoint [MDT07, TMD09, CBB+10]. Clements et al. [CBB+10] suggest the documentation of a system architecture from overlapping and interrelated viewpoints describing, e.g., the system's modules, its components and connectors, the allocation of software components to non-software structures, and the system's behavior.

Similarly, Medvidovic et al. [MDT07] have observed the extension of ADLs from the more technical side elaborated in their classification framework in [MT00] to application domain-specific descriptions. Furthermore, some ADLs support or reflect the business context by adapting to processes (e.g., the ADL ArchiMate [The12]), supporting product lines (e.g., Koala [vOvdLKM00] and MontiArc [HRRS11, HRR+11]), or allowing cost analyses [MDT07]. A recent survey on the use of architectural languages in industry [MLM+13] shows that the five most useful features rated by industrial ADL users are the support for iterative architecting, well-defined semantics, tool support, checking the alignment between an architecture model and the implementation, and the extensibility of the ADL.

These features and characterizations show that architecture descriptions and their applications go far beyond C&C models. Despite these extensions most ADLs contain means to describe C&C models or even focus descriptions based on components, ports, and connectors. Thus, we consider C&C models a common central subset of architecture descriptions. In the following we describe some ADLs from the literature and describe their modeling capabilities for C&C models.

The Architecture Analysis Design Language (AADL) [wwwa, FGH06, FG12] is an architecture description language. AADL is standardized by the Society for Automotive Engineers. AADL models contain component type declarations and implementation declarations. Component type declarations define the features of a component including directed and typed ports. Implementation declarations define the internal structure of a component in terms of subcomponent instances and connectors. A special design concept of AADL are predefined component categories. These are categories of application software, e.g., `thread` or `subprogram`, the execution platform, e.g., `memory`, composite, and generic components with the categories `system` and `abstract`. This domain-specific orientation of AADL makes it suitable for embedded realtime systems especially in the automotive, avionics, and aerospace domains [wwwa, HWF+10, BCK+11, FG12].

Acme [GMW00, SG04] is an ADL that was initially developed as an architecture language interchange format [GMW97] containing the common elements of various ADLs.

These elements comprise components, ports and connectors. Acme has evolved to a stand alone ADL. One of the main features of Acme and its tool support AcmeStudio [wwwy] is the definition and analysis of architectural styles [SG04, KG10]. Our approach to C&C model synthesis supports architectural styles for C&C models as presented in Section 5.5.

ArchiMate [LPJ10, JPL+11] is a modeling language for enterprise architecture. In a recent survey on the use of ADLs in industry [MLM+13] it was reported the second most used ADL after UML [Obj12a]. ArchiMate divides the description of a system into three layers: the business layer, the application layer, and the technology layer. The application layer contains models of the application software that are organized in components. Components can interact via provided and required application interfaces or via application collaboration [The12].

MontiArc [HRR12] is an ADL designed for modeling cyber-physical systems [Lee08]. The ADL is developed based on the core ADL concepts identified in [MT00] and its semantics is based on the stream processing theory FOCUS [BS01]. MontiArc is a textual ADL developed using the language workbench MontiCore [KRV10]. One important feature of MontiArc is its typing and instantiation mechanism for components. The structure of composed components is defined as the instantiation of components and the connection of the ports of their interfaces. MontiArc supports the definition of generic component types that can be instantiated with type parameters to *configure*, e.g., the type of a port.

The ADL MontiArc is developed in an extensible way. We have implemented an extension via stereotypes for C&C views presented in Section 3.6. Another extension of MontiArc is the language MontiArcAutomaton [RRW12, RRW13b], which embeds automata in atomic components. Haber et al. [HRRS11, HRR+11, HRRS12] have extended MontiArc with support for modeling variability in the context of software product lines.

Many works also consider the Unified Modeling Language (UML) [Obj12a] an architecture description language [ICG+04, MDT07, MLM+13]. The UML consists of 14 diagram types for modeling the structure and the behavior of software systems. Ivers et al. [ICG+04] describe how to use UML 1.4 and UML 2.0 for documenting component and connector models. One major criticism by Ivers et al. is that UML connectors are not first class elements. In UML it is thus not easily possible to associate behavior and interface descriptions [ICG+04]. In our work we also use a basic model of connectors. Our connectors do not exhibit interfaces, behavior, or roles.

The Systems Modeling Language (SysML) [FMS11, Obj12b, Wei07] is a derivation of UML 2 defined as a profile using UML's profile mechanism [Obj12a]. SysML introduces block definition diagrams (based on UML class diagrams) and internal block diagrams (based on UML composite structure diagrams). It extends ports and flows of UML and thus overcomes some of the criticized weaknesses of UML for the definition of component and connector models.

A variety of tools implements C&C models similar to the structure defined in Definition 2.2. One popular example is the block diagram language implemented in MathWorks Simulink [wwwn]. Block diagrams consist of blocks and lines. One kind of the blocks in block diagrams are function blocks, which have input and output signals that are con-

nected via lines. Multiple blocks can be composed to subsystems with input and output ports. Subsystems are again a special kind of blocks. In the terminology of C&C models blocks are components, signals and ports correspond to ports, and lines correspond to connectors.

Another implementation of C&C models is provided within the AutoFOCUS tool suite [BHS99, HF07, wwwe]. C&C models are modeled as component architectures consisting of components with typed input and output ports that are connected via channels (connectors). Components in C&C architectures can be composed and consist of further components or they are atomic and their behavior is specified using behavioral models or code implementations.

As a concrete ADL for C&C models we use MontiArc [HRR12]. We consider supporting the subsets of other ADLs and related formalisms that match the structure of C&C models as defined in Definition 2.2 a technical issue.

Chapter 3.

Component and Connector Views for Component and Connector Models

Existing languages and tools for C&C modeling are built around the implementation-oriented hierarchical decomposition of systems to subsystems. This hierarchical decomposition supports a distributed development process of components with well-defined interfaces and fits well with downstream code generation. However, it limits possible support for specifying structural properties that reflect the partial knowledge available to different stakeholders and their interests at different stages of the system's development life-cycle, which inevitably crosscut the boundaries of subsystems.

We present *C&C views*, as a language for C&C modeling that enables the specification of structural properties that crosscut the boundaries of subsystems. C&C views take advantage of novel abstraction mechanisms for hierarchy and connectivity, not present in comparable languages. These mechanisms enable different stakeholders to create views that express partial knowledge about the structure of the system at hand, corresponding to different use cases, functions, or concerns.

A C&C view documents one or more design decisions and their relevant environment. C&C views are independent of C&C models and can be reused, modified, and evolved independently of a concrete C&C model. Different analyses connect C&C views with C&C models. We formally define the semantics of C&C views by a satisfaction relation between C&C models and C&C views.

Moreover, we lift the definition of single C&C views to C&C views specifications. A C&C views specification consists of a propositional formula over C&C views. Thus, a specification may contain, e.g., positive views, which should be satisfied, negative ones, which should not be satisfied, disjunctions between views, allowing one to express alternative designs, and implications, allowing one to express dependencies between design decisions.

We introduce the three main applications supported by our work. First, documentation, to highlight and make knowledge contained in a C&C model explicit. Second, verification, deciding whether a C&C model satisfies a C&C view. Third, synthesis, automatically constructing a satisfying C&C model given a C&C views specification. We have implemented C&C views verification and synthesis of C&C models. We present C&C views verification in Chapter 4 and synthesis from C&C views specifications in Chapter 5.

Chapter outline and contributions

Section 3.1 presents a pump station system and its C&C model provided by the AutoFOCUS IDE [wwwd]. We use the pump station as a running example in Chapter 3 and Chapter 4. Section 3.2 continues with an overview of C&C views usage scenarios. A formal definition for C&C views is given in Section 3.3 and for C&C views satisfaction in Section 3.4 expressing C&C views semantics in relation to C&C models. The formalization of C&C views, their abstraction mechanisms over C&C models, and their semantics constitute a main contribution of this thesis.

Another contribution presented in this chapter is the extension of C&C views to C&C views specifications in Section 3.5 allowing one to express mandatory, alternative, and negative structural properties using C&C views.

We introduce a language profile of the ADL MontiArc for modeling C&C views in Section 3.6. Finally, Section 3.7 discusses variations and extensions of C&C views and related approaches for specifying structural properties of C&C models and architectural models in general.

3.1. Introducing the Pump Station System

We use the pump station system C&C model given with the AutoFOCUS IDE [BHS99, HF07, wwwd] as a starting point for our running example.

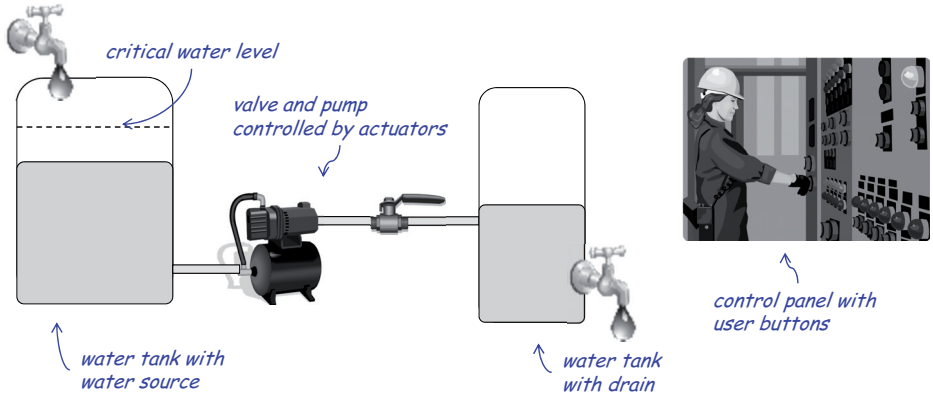


Figure 3.1.: An illustration of the pump station with two water tanks, a pump, a valve, and a control panel.

The physical structure and parts of the pump station are illustrated in Figure 3.1. The pump station consists of two water tanks, a pump, a valve, and a control panel. Water flows into the first tank and the water level rises. The water level can be pumped to the second water tank, which has a drain, by the pump if the valve is opened. The pump becomes active when either an engineer activates it via the control panel, or the

water level in the first tank rises to a critical level.

A team of engineers has modeled the software C&C model of the pump station system consisting of components and connectors and their composition at different containment levels. The C&C model of the pump station includes the pumping system and a model of the environment for simulation, as shown in Figure 3.2 using separate subfigures for each component definition. Components are either atomic and not further detailed here or they are defined by composition of subcomponents and connectors.

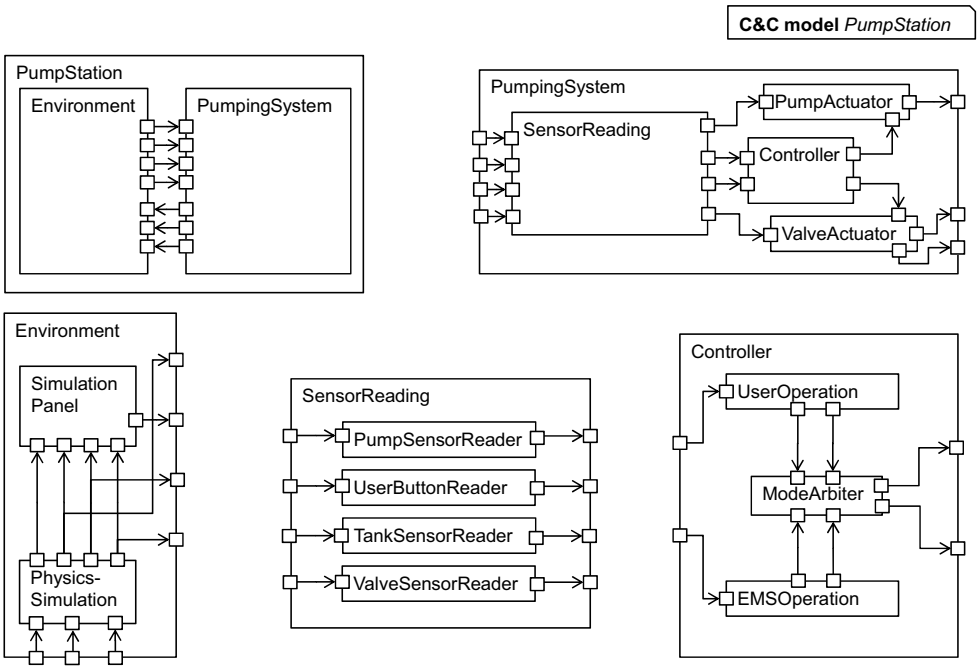


Figure 3.2.: The C&C model of the pump station system adapted from an example of the AutoFOCUS IDE [wwwd]. The C&C model is shown in five separate component definitions as presented by state-of-the-art C&C modeling languages and tools, e.g., AcmeStudio [wwwy] and AutoFOCUS. To omit clutter we do not show port names or data types in the figure. The complete model with port names and data types is shown in Appendix G.

The root of the pump station C&C model is the component `PumpStation` with its two subcomponents `Environment` and `PumpingSystem`. The logical architecture of the software system to control the pump station (shown in Figure 3.1) is contained in the component `PumpingSytem`. The component `PumpingSystem` consists of the component `SensorReading` for reading sensor data from different sources, a controller component `Controller`, and actuators for effecting changes on the pump (component

PumpActuator) and on the valve (component ValveActuator).

The component SensorReading has subcomponents that read data from various sensors, e.g., the pump sensor, a button on the control panel, the tank sensor, and the valve sensor. The component Controller receives the inputs from the user button sensor and the tank sensor. Internally, the component Controller independently computes desired control commands for the valve and the pump in its subcomponents UserOperation and EMSOperation. The component ModeArbiter decides based on its specification which commands to forward to the pump and valve actuators.

The second subsystem of the pump station C&C model is modeled as the component Environment. It is used by the engineers for the simulation of the pump station. The component Environment consists of two subcomponents: a simulation panel (component SimulationPanel) to simulate the control panel of the pump station and a physics simulation (component PhysicsSimulation) to compute the water levels of the tanks.

3.2. Component and Connector Views Usage Scenarios and Language Features

We now present usage scenarios for C&C views during the development and maintenance of C&C models. In Section 3.2.2 we highlight the key language features of C&C views supporting these usage scenarios before we formally define the structure of C&C views in Section 3.3.

3.2.1. Usage Scenarios for C&C Views

We present examples of C&C views and various usage scenarios in the context of the pump station system. The presentation of C&C views in this section is semi-formally. Formal definitions are given in Section 3.3.

Documentation

C&C views can be used to document partial knowledge about a C&C model. Consider the following scenario.

The team wants to create documentation for the C&C model of the pump station, to make knowledge contained in the model explicit and to guide new team members in understanding the C&C model design. One important information is that the pump sensor reader is connected to the pump actuator via the actuator's incoming port pumpState and that the valve sensor reader is connected to the valve actuator via the port valvePosition.

The team documented this information by creating the C&C view ASPumpingSystem, shown in Figure 3.3 (a). Please note that the view abstracts away the (direct) containment of the sensors inside component SensorReading, which is not shown in the view,

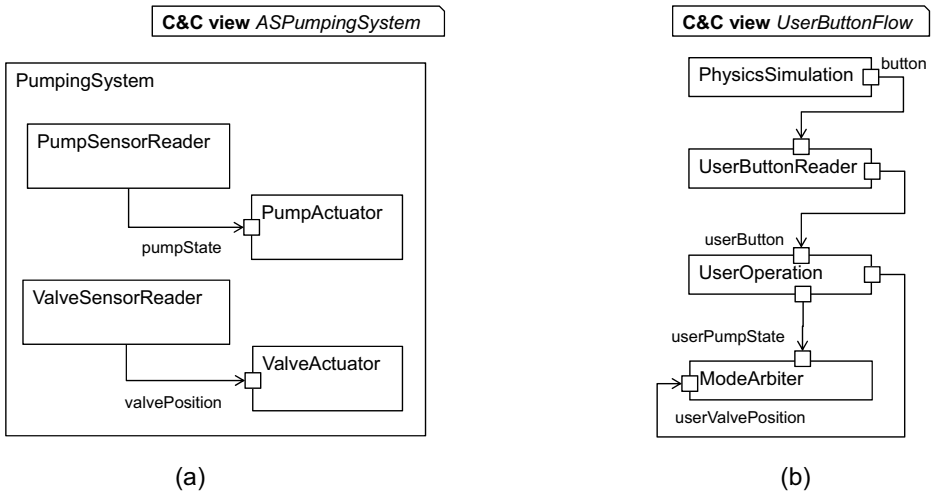


Figure 3.3.: Two C&C views related to the pumping station system. The C&C view `ASPumpingSystem` documents the C&C model `PumpingSystem` by showing relevant components and connectors across different containment levels. The C&C view `UserButtonFlow` shows components and connectors participating in the flow of user button messages.

and does not mention the exact source ports of the connectors from the sensors, although this information appears in the C&C model.

A team member created another view, `UserButtonFlow`, shown in Figure 3.3 (b), to document the components and connectors handling the pressing of the user button. The engineer decided to document all relevant ports but only some port names. She abstracted away parent components, e.g., `Environment` and `SensorReading`, and presented all relevant components at a single level.

C&C views verification

A C&C view can be used as a specification that a C&C model should satisfy. Satisfaction can be checked automatically. Consider the following scenario.

A team of engineers has described its design in several C&C views, highlighting relevant design decisions for the C&C model. The team is using the views as specifications and wants to automatically check whether the C&C model they built satisfies these views.

The C&C model `PumpStation` depicted in Figure 3.2 satisfies the C&C view `ASPumpingSystem` shown in Figure 3.3 (a) (formally written as `PumpStation` $\vDash$ `ASPumpingSystem`), since all components shown in the view exist in the C&C model, the model preserves the containment relation between components in the view, and the ports and connectors in the view are concretized in the definitions of components

`SensorReading` and `PumpingSystem` (connectors in the view have corresponding chains of connectors in the model).

When checking the model against the view `UserButtonFlow`, the team discovers a mismatch between the C&C view specification and the pump station C&C model (formally written as $\texttt{PumpStation} \not\models \texttt{UserButtonFlow}$). There is no chain of connectors from component `PhysicsSimulation` to component `UserButtonReader`. When replacing component `PhysicsSimulation` by component `SimulationPanel`, the C&C view is satisfied by the pump station model.

In the set of C&C views, used as a specification for the C&C model, the team formalized both desired and undesired designs: positive C&C views, which should be satisfied, and negative C&C views, which should *not* be satisfied by the C&C model they build. Throughout the system's design the engineering team applies C&C views verification to check that the C&C model satisfies the positive views and does not satisfy the negative ones.

C&C views synthesis

A C&C views specification consisting of positive and negative views can be used not only for verification, i.e., after a C&C model is built, but also as input for a synthesis process that outputs a correct by construction C&C model, one which satisfies the positive views and does not satisfy the negative ones, if such a C&C model exists. Consider the following scenario.

An engineering team is going to develop the software C&C model of a pump station. The team has no initial C&C model, but different team members have partial knowledge available.

One team member knows that the pumping system will contain a pump actuator and a valve actuator that are both connected to relevant sensors. She expresses this knowledge as the C&C view `ASPumpingSystem` depicted in Figure 3.3 (a). Another team member describes a component `EmergencyController` that is part of the pump station and is connected to component `UserOperation`, which is further connected to component `ModeArbiter`. The engineer currently cannot specify further properties of the connection, e.g., source or target port name and type, between the emergency controller and component `UserOperation`. She simply specifies that these two are connected. The C&C view `SystemEmergencyController` created by the engineer is shown in Figure 3.4 (a).

The C&C view `SystemEmergencyController` from Figure 3.4 (a) needs only to be satisfied if the component `EmergencyController` is indeed part of the system. The existence of the component `EmergencyController` is expressed in the C&C view `ExistsEmergencyController` from Figure 3.4 (b). The dependency of the views is formalized as $(\texttt{ExistsEmergencyController} \Rightarrow \texttt{SystemEmergencyController})$ inside a C&C views specification. An architect combines the C&C views to a views specification, i.e., a propositional formula over the views, which may contain positive, negative, alternative, and dependent views. The architect uses our tool to synthesize a

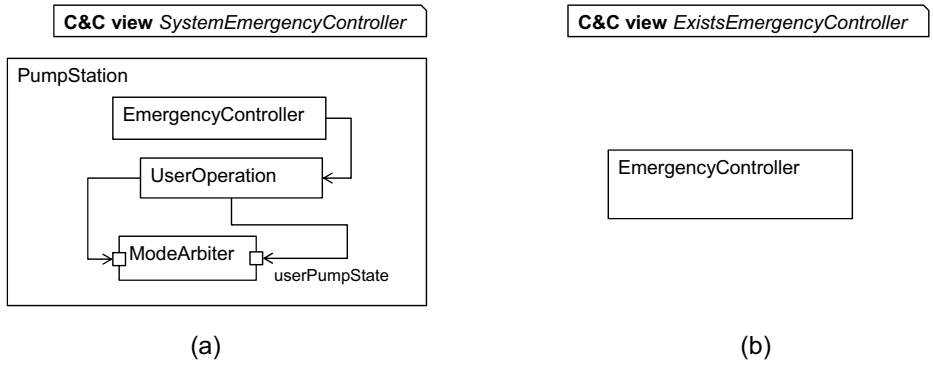


Figure 3.4.: Two C&C views related to the pumping station system. The views document relations between component `EmergencyController` and its relevant part of the pumping system.

satisfying C&C model. The synthesized C&C model is complete (all ports have data types and names) and is ready for code generation.

3.2.2. C&C Views Key Language Features

The key, distinctive language features of C&C views include three abstraction mechanisms: an abstraction of hierarchy, an abstraction of connectivity, and an abstraction of interfaces.

The syntax we use for C&C views is intentionally taken from C&C models. No special syntactic constructs are added, so as to keep the language simple and ensure ease of use and learning: any engineer familiar with C&C models can specify C&C views.

Hierarchy abstraction

Putting one component inside another in a view does not mean that the second is necessarily a direct subcomponent of the first. Rather, it means that the second is contained in the first, but not necessarily directly. Putting two components as siblings in a view does not necessarily mean that they are direct subcomponents of a common parent, but it means that they are not contained within one another. This abstraction allows engineers to specify partial knowledge about the hierarchical structure of the system they build.

As an example consider the components `PumpSensorReader` and `PumpActuator` in the view `ASPumpingSystem` shown in Figure 3.3 (a). These two components are not contained within one another but they are not necessarily sibling components in a satisfying C&C model. As another example consider component `PumpStation` in view the `SystemEmergencyController` shown in Figure 3.4 (a) and the three components shown inside it, e.g., `ModeArbiter`. These components need to be contained in

component PumpStation in any satisfying C&C model, but not necessarily as direct subcomponents.

Connectivity abstraction

Connectors appearing in a view are abstract: connecting two components in a view with a connector does not mean that they are directly connected with a single connector. Rather, it means that they are connected by a chain of connectors (all having the same data type), leading from one component to the other. As opposed to connectors in C&C models, an abstract connector in a view may connect components directly via optional ports.

As an example consider the abstract connector between component UserButtonReader and component UserOperation shown in view UserButtonFlow in Figure 3.3 (b). This abstract connector is implemented by a chain of three connectors as shown in view UserButtonFlowImpl in Figure 3.5. The view UserButtonFlow with component PhysicsSimulation renamed SimulationPanel is satisfied by the C&C model. Its complete implementation by the C&C model PumpStation is shown in Figure 3.5.

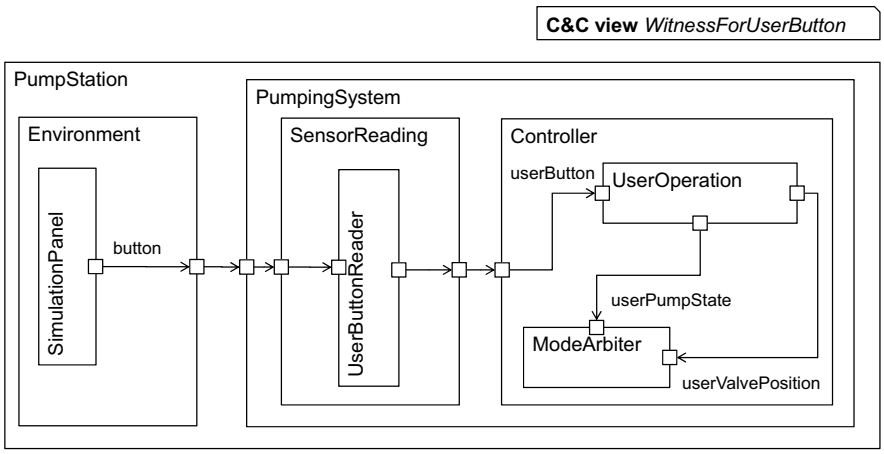


Figure 3.5.: A C&C view generated to show the implementation of the view UserButton (view UserButtonFlow from Figure 3.3 with component PhysicsSimulation renamed SimulationPanel) in the C&C model PumpStation shown in Figure 3.2.

Incomplete interfaces

The third abstraction mechanism in C&C views concerns component interfaces, consisting of port names and their types. Specifically, component interfaces presented in a view

may be incomplete, and not show all port names and all types.

As an example consider the view `SystemEmergencyController` in Figure 3.4 (a). The interface of component `EmergencyController` is not given at all, although the component is the source of an abstract connector. For component `ModeArbiter` two ports are shown of which only one is named.

Extension points for additional language features

Variability within the definition of a modeling language allows the extension and customization of a modeling language's syntax and semantics [CGR09, GR10]. Our modeling language for C&C views contains an extension mechanisms to attach additional properties to components, ports, subcomponents, and connectors via stereotypes (see Section 3.6.5).

We have used this extension mechanism of our modeling language for C&C views to specify further knowledge available to the engineers and add it to the views. We present two examples for these extensions. An engineer can specify that a component in a view is atomic, i.e., the component may not have subcomponents or internal connectors in any satisfying C&C model. Engineers can further specify that their knowledge of a components interface is complete, i.e., the component has exactly the ports specified in the C&C view in any satisfying C&C model.

3.3. Component and Connector Views

We give a formal definition of the structures of C&C views with well-formedness rules for valid views. C&C views consist of components at different containment levels, directed, possibly typed, and possibly named ports, and connectors connecting components or ports.

Definition 3.6 (Component and connector view)**.** A component and connector view is a structure $view = (Cmps, Ports, PNames, AbsCons, Types, subs, ports, dir, type, name, stereotypes)$ where

1. $Cmps$ is a set of components $cmp \in Cmps$ (with unique names), each of which has a set of ports $ports(cmp) \subseteq Ports$ and a set of subcomponents $subs(cmp) \subset Cmps$,

2. $Ports$ is a set of directed input and output ports $p \in Ports$ with $dir(p) \in \{IN, OUT\}$ where each port has a (possibly unknown) name $name(p) \in PNames \cup \{\bot\}$[1], a (possibly unknown) type $type(p) \in Types \cup \{\bot\}$, and belongs to exactly one component $\exists_! cmp \in Cmps : p \in ports(cmp)$,

3. $AbsCons$ is a set of directed abstract connectors $con \in Cons$, each of which connects two components optionally via their ports $(con.srcPort \in ports(con.srcCmp) \vee$

[1] We use $\bot$ to denote unknown names, types, and ports.

$con.srcPort = \bot$ and $con.tgtPort \in ports(con.tgtCmp) \vee con.tgtPort = \bot$), connected ports have the same or an unknown type ($|\{type(con.srcPort), type(con.tgtPort)\} \setminus \{\bot\}| \leq 1$), and

4. $Types$ is a finite set of types $t \in Types$ that appear on ports: $t \in Types \Leftrightarrow \exists p \in Ports : type(p) = t$.

Additionally, the following rules for well-formedness apply:

5. $\nexists c \in Cmps : c \in subs^+(c)$, where $subs^+$ denotes the transitive closure of the subcomponent relation $subs : Cmps \times Cmps$, i.e., no component is its own (transitive) parent,

6. $\forall child \in Cmps : |\{parent \in Cmps \mid child \in subs(parent)\}| \leq 1$, i.e., every component has at most one direct parent in the view, and

7. $\forall cmp \in Cmps : \forall p_1, p_2 \in ports(cmp) : \bot \neq name(p_1) = name(p_2) \Rightarrow p_1 = p_2$, i.e., known port names are unique within each component.

Variability mechanism: Stereotypes may be added to the elements of the sets $Cmps$, $Ports$, and $AbsCons$ to extend or modify the semantics of C&C views. For example, the set of stereotypes added to a component $cmp \in Cmps$ is the set $stereotypes(cmp)$. △

Please note that in a C&C view, abstract connectors are not restricted to the four cases of connectors in C&C models as defined in Definition 2.2, Item 9. Rather, they may connect between any two ports and components in the view. Also, ports may have multiple incoming connectors in C&C views since these might refer to the same concrete chain of connectors in a C&C model. We do not restrict C&C views to have exactly one top component.

Notation: For a C&C view $view$, a port $p \in view.Ports$, and a component $c \in view.Cmps$ we use the short notation $p \in c.ports$ to denote $p \in view.ports(c)$ in case the C&C view $view$ is clear from the context. In addition, we write $view.name$ and $c.name$ (for $c \in Cmps$) for the unique names of views and components.

C&C models and C&C views share the same structural elements, i.e., hierarchically composed components as well as directed, named, and typed ports. They however differ in the amount of required information specified about these elements and in their semantics.

Because of the common structural elements between C&C views and C&C models we are again able to use the modeling language MontiArc to model C&C views as demonstrated in Section 3.6.

3.3.1. Language Variability Mechanism

We have added a language extension mechanism based on stereotypes to the C&C views modeling language. We follow the framework described in [CGR09] to define variability for C&C views.

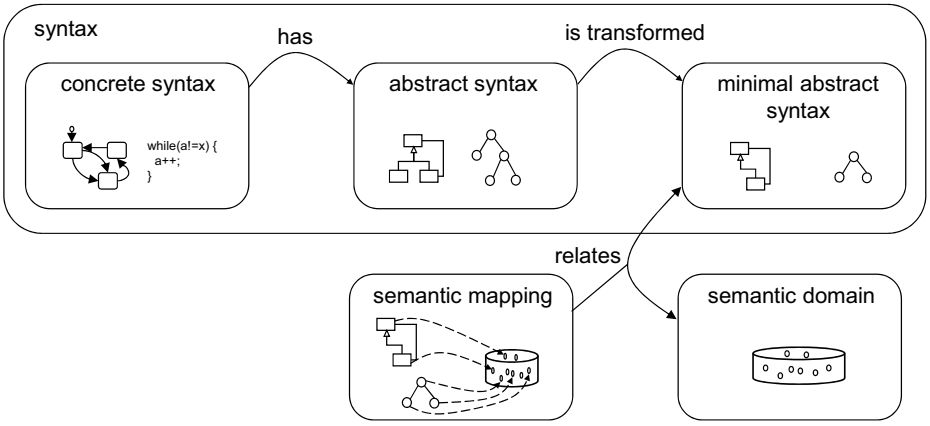


Figure 3.7.: The basic parts of a modeling language definition in the framework defined in [CGR09]. We define the concrete and abstract syntax based on MontiArc in Section 3.6. The minimal abstract syntax for C&C views is the structure given in Definition 3.6.

Figure 3.7 shows the conceptual parts of a modeling language definition as presented in [CGR09]. For us, the concrete and abstract syntax are defined by a MontiCore grammar (the grammar of the modeling language MontiArc, see Section 3.6). The minimal abstract syntax is the structure of C&C views given in Definition 3.6.

According to the classification presented in [CGR09, Table 1] modeling language variability is classified as presentation variability, syntactic variability, and semantic variability. Presentation variability refers to variability on the presentation of elements in graphical or textual concrete syntax. We present our textual syntax for C&C views in Section 3.6. Syntactic variability is based on stereotypes, language parameters and context conditions (well-formedness rules) [CGR09]. We define syntactic variability for C&C views based on stereotypes and add additional rules for the well-formedness of C&C views with modified abstract syntax.

Stereotypes may be applied to the key elements of C&C views: components, ports, and abstract connectors. We give two examples of variations for C&C views and informally sketch the semantics of the language extension. A formal definition of the semantics of the extensions in terms of a satisfaction relation between C&C models and C&C views is presented in Section 3.4.1.

Atomic components An atomic component is a component that is not further decomposed, i.e., that has no subcomponents. In case an engineer knows that a component modeled in a C&C view is an atomic component in any C&C model that satisfies the view, this knowledge can be attached to the component in the view using the stereotype «`atomic`». Expressing the atomicity of a component without this language extension is not easy and might require multiple C&C views.

In addition to the well-formedness rules described in Definition 3.6 this language extension adds the new well-formedness rule:

$$\forall cmp \in Cmps : \text{«atomic»} \in stereotypes(cmp) \Rightarrow subs(cmp) = \varnothing$$

Interface-complete components In case an engineer knows that the interface of a component specified in a C&C view is the complete interface of the component in every satisfying C&C model, this knowledge can be attached to the component in the view using the stereotype «interfaceComplete».

In addition to the well-formedness rules described in Definition 3.6 the language extension adds the new well-formedness rule:

$$\forall cmp \in Cmps : \text{«interfaceComplete»} \in stereotypes(cmp) \Rightarrow \forall p \in ports(cmp) : name(p) \neq \bot$$

The rule above requires that all ports of an interface-complete component have names. It is not required that ports have known types. This gives more freedom to the engineer creating the view with the interface-complete component.

The extension mechanism of C&C views, which is based on syntactic variability, also requires semantic variability to assign a meaning to the extended language, i.e., the stereotypes attached to the components, ports, and abstract connectors. For the two above examples we show how syntactic variability influences the semantics of C&C views in terms of their satisfaction by C&C models in Section 3.4.1.

Unless stated otherwise by C&C views we refer to the basic C&C views from Definition 3.6 without an extension with stereotypes.

3.4. Satisfaction of Component and Connector Views

Based on the formal definitions of the structures of C&C models from Definition 2.2 and C&C views from Definition 3.6 we now define the satisfaction of a C&C view by a C&C model.

Roughly, a C&C model satisfies a C&C view if and only if the types, components, and ports mentioned in the view are contained in the C&C model, the C&C model respects the subcomponent relation induced by the view, two ports connected by an abstract connector in the view are connected by a chain of connectors in the C&C model (respecting direction, names, and types), and all ports of a component in the view belong to the same component in the C&C model with corresponding name, type and direction. More formally:

Definition 3.8 (C&C model $\vDash$ C&C view)**.** A C&C model m satisfies a C&C view *view* if and only if:

1. $view.Types \subseteq m.Types$,
2. $view.Cmps \subseteq m.Cmps$,

3. $\forall cmp_1, cmp_2 \in view.Cmps$:
   $cmp_1 \in view.subs(cmp_2)$ if and only if $cmp_1 \in m.subs^+(cmp_2)$
   (we use $^+$ to denote the transitive closure),

4. $\forall cmp \in view.Cmps$:
   (a) $\forall p \in view.ports(cmp)\ \exists p' \in m.ports(cmp) : p \cong p'$ where
   (b) $p \cong p'$ if and only if
   (b1) $view.dir(p) = m.dir(p') \wedge$
   (b2) $view.type(p) \in \{\bot, m.type(p')\} \wedge$
   (b3) $view.name(p) \in \{\bot, m.name(p')\}$, and

5. $\forall ac \in view.AbsCons\ \exists c_1, ..., c_n \in m.Cons$ such that
   (a) $ac.srcCmp = c_1.srcCmp \wedge (ac.srcPort \cong c_1.srcPort \vee ac.srcPort = \bot) \wedge$
   (b) $ac.tgtCmp = c_n.tgtCmp \wedge (ac.tgtPort \cong c_n.tgtPort \vee ac.tgtPort = \bot) \wedge$
   (c) $\forall 1 \leq i < n : c_i.tgtPort = c_{i+1}.srcPort$.

△

Definition 3.8 formally defines the satisfaction relation between a C&C model and a C&C view. For the data types on ports and for components we use the same structures in C&C models and C&C views which allows us to use set inclusion in Item 1 and Item 2.

The matching of ports is more complicated since we can not identify ports by their name, which is optional in C&C views. For a given C&C model m and a C&C view $view$ we thus introduce the relation $\cong \subseteq (view.Ports \times m.Ports)$ for matching ports in Item 4 (b).

Please note that a well-formed C&C view with a satisfying C&C model m has infinitely many satisfying C&C models because the set $m.Cmps$ and the subcomponent function $m.subs$ can always be extended with additional subcomponents without affecting satisfaction. Also note that by definition the empty view is satisfied by any C&C model.

3.4.1. Language Variability and Satisfaction

The language variability mechanism introduced in C&C views in Definition 3.6 also has an impact on the semantics of C&C views captured by the satisfaction relation of C&C models and C&C views. We show how to adapt the satisfaction relation defined in Definition 3.8 to support the two example language extensions for atomic components and interface-complete components.

Atomic components A C&C model m satisfies a C&C view $view$ extended with the stereotype «`atomic`» for components if and only if the C&C model m satisfies the C&C view $view$ according to Definition 3.8 (or its extensions for further language variations) and

$$\forall cmp \in view.Cmps : \text{«atomic»} \in cmp.stereotypes \Rightarrow m.subs(cmp) = \varnothing$$

Interface-complete components A C&C model m satisfies a C&C view $view$ extended with the stereotype «`interfaceComplete`» for components if and only if the

C&C model m satisfies the C&C view *view* according to Definition 3.8 (or its extensions for further language variations) and

$$\forall cmp \in view.Cmps : \text{«}\texttt{interfaceComplete}\text{»} \in cmp.stereotypes \Rightarrow \\ \forall p \in m.ports(cmp)\ \exists p' \in view.ports(cmp) : m.name(p) = view.name(p')$$

Please note that these two language extensions exhibit nice properties with respect to the satisfaction relation. First, the two extensions are independent, i.e., they do not require or contradict each other. Second, the language extensions refine the semantics of C&C views, i.e., they strengthen the satisfaction relation by additional constraints. These two properties do not necessarily hold for all possible variations of C&C views. In general, variations of the syntax and semantics of modeling languages may influence or contradict each other. A more sophisticated handling of language variability is proposed in [Grö10, GR10] using feature models [CE00] to formalize and manage dependencies between different variations.

3.5. Component and Connector Views Specifications

A C&C views specification combines multiple views in a single propositional formula. A C&C views specification allows to define positive views, which should be satisfied, negative views, which should not be satisfied, and dependent views, where the satisfaction of one view implies the satisfaction of another.

Definition 3.9 (C&C views specification)**.** A C&C views specification S is a propositional formula over a set of C&C views V. △

By natural extension of Definition 3.8, a C&C model m satisfies a specification S, denoted $m \vDash S$, if and only if for each view $v \in V$ the view name replaced by the Boolean valuation $m \vDash v$ satisfies the propositional formula S. We denote replacing each $v \in V$ in S by the valuation of $m \vDash v$ as $S[v/(m \vDash v)]_{v \in V}$ in Definition 3.10.

Definition 3.10 (C&C model $\vDash$ C&C views specification)**.** A C&C model m satisfies a C&C views specification S over a set of views V if and only if $S[v/(m \vDash v)]_{v \in V}$. △

Consider a views specification S_1 over the views `ASPumpingSystem`, `UserButtonFlow` (shown in Figure 3.3 (b)), `UserButtonRemote` (shown in Figure 3.11 (a)), `SystemEmergencyControllerFixed` (shown in Figure 3.11 (b)), `EmergencyInsidePumpingSystem` (shown in Figure 3.11 (c)), and `UserButtonReaderInsideController` (shown in Figure 3.12). The view `UserButtonRemote` is similar to the view `UserButtonFlow` (shown in Figure 3.3 (b)) but the specified data flow path for the user button does not start at component `SimulationPanel` but at component `RemoteControlPanel`. The view `EmergencyInsidePumpingSystem` specifies the existence of component `EmergencyController` inside component `PumpingSystem`.

Equation 3.1 shows the propositional formula for the C&C views specification S_1.

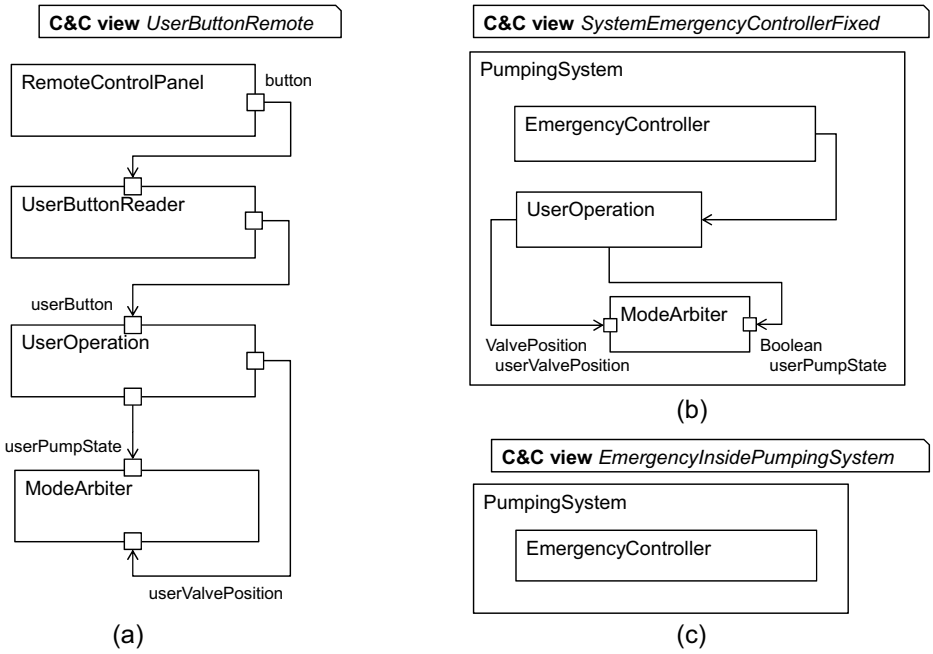


Figure 3.11.: Three C&C views used in specification S_1: `UserButtonRemote` (a), `SystemEmergencyControllerFixed` (b), and `EmergencyInsidePumpingSystem` (c).

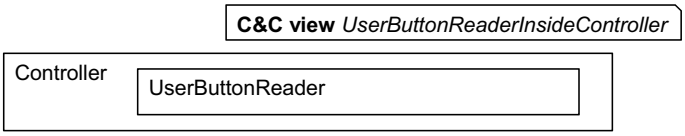


Figure 3.12.: The C&C view `UserButtonReaderInsideController` that depicts the component `UserButtonReader` contained in the component `Controller`.

$$S_1 = \texttt{ASPumpingSystem} \wedge \neg \texttt{UserButtonReaderInsideController} \wedge (\texttt{EmergencyInsidePumpingSystem} \Rightarrow \texttt{SystemEmergencyControllerFixed}) \wedge (\texttt{UserButtonFlow} \vee \texttt{UserButtonRemote}) \quad (3.1)$$

S_1 specifies that the C&C model should satisfy the view `ASPumpingSystem`, that it should not satisfy the view `UserButtonReaderInsideController`, that if it satisfies the view `EmergencyInsidePumpingSystem` then it should also satisfy the view

SystemEmergencyControllerFixed, and that it should satisfy the UserButtonFlow view or the UserButtonRemote view.

It is easy to check whether the C&C model PumpStation introduced in Section 3.1 satisfies or does not satisfy each of the C&C views mentioned in S_1. Specifically, the C&C model satisfies the view ASPumpingSystem and the view UserButtonFlow but it does not satisfy the view the view UserButtonRemote, the view UserButtonReaderInsideController, the view EmergencyInsidePumpingSystem, and the view SystemEmergencyControllerFixed. Assigning the resulting truth values to the corresponding variables in the propositional formula shown in S_1 (Equation 3.1) shows that PumpStation $\models S_1$.

3.6. Modeling Component and Connector Views using MontiArcView

We now present a language profile of the ADL MontiArc for documenting C&C views. The language profile MontiArcView extends the MontiArc language with several stereotypes.

3.6.1. Defining a View

Every MontiArcView C&C view definition is marked with the stereotype «view» on the top level. While in C&C models the first component definition is the root of the system, the stereotype «view» around the outer definition does not resemble a component but a view, which is a collection of the components of interest and the relations known between them. Line 3 in Listing 3.13 defines the view UserButton.

3.6.2. Containment and Independence of Components

Listing 3.13 shows the view UserButton which contains the two components UserButtonReader and SimulationPanel. In a view the knowledge specified about a C&C model is partial. For example, the component UserButtonReader does not necessarily need to be a sibling component of component SimulationPanel. The two components are however shown side by side meaning that they are not contained in each other in any satisfying C&C model.

The component Sensor (l. 10) is shown to be contained in component UserButtonReader. This containment is not necessarily concrete — component Sensor might be a direct subcomponent of UserButtonReader or contained in any of its subcomponents that might exist but are not shown inside the view.

3.6.3. Component Interface

In C&C views the interfaces of components may be underspecified. The interface of a component can be either completely omitted, contain untyped or unnamed ports, or can be marked as complete.

MontiArcView

```
1  package pumpStationExample;
2
3  <<view>> component UserButton {
4
5    component UserButtonReader {
6      port
7        <<untyped>> in button,
8        <<unnamed>> out UserInput;
9
10     component Sensor {
11     }
12   }
13
14   component SimulationPanel {
15     port
16       out Button button;
17   }
18 }
```

Listing 3.13: A view of the system called `UserButton`.

Missing Interface If a component in a view has no definition of an interface (starting with keyword `port`) nothing is known about the component's interface (see, e.g., component `Sensor` in line 10 of Listing 3.13). It is not restricted nor required to contain any ports in a satisfying C&C model.

Untyped Port The stereotype «`untyped`» in front of the port of a component, e.g., the first port of component `UserButtonReader` in Listing 3.13 indicates that the type of the port with the name `button` (Listing 3.13, l. 7) is not known in the view. In a satisfying C&C model there will be an incoming port with name `button` and a concrete type.

Unnamed Port The stereotype «`unnamed`» on the second port of component `UserButtonReader` means that the name of the port is not known in the view. The C&C view specifies that the port has the direction outgoing and the type `UserInput` (Listing 3.13, l. 8).

In all cases (unnamed or untyped) the direction of the port has to be specified inside C&C views.

3.6.4. Abstract Connectors

Abstract connectors allow the abstraction of connectedness between components and ports in C&C models.

In views connections are not defined on the level of the containing component but the level of the view. Listing 3.14 shows the definition of three abstract connectors in the C&C view `UserButtonWithConnections`.

MontiArcView

```
1  package pumpStationExample;
2
3  <<view>> component UserButtonWithConnections {
4
5    component UserButtonReader {
6      port
7        in Button button;
8    }
9
10   component SimulationPanel {
11     port
12       out Button button;
13
14     component PreProcessor {
15       port
16         <<untyped>> out trans;
17     }
18   }
19
20   component Environment {
21   }
22
23   connect SimulationPanel -> Environment;
24   connect UserButtonReader -> SimulationPanel.button;
25   connect PreProcessor.trans -> Environment;
26 }
```

Listing 3.14: The C&C view `UserButtonWithConnections` in MontiArcView syntax. The abstract connectors in lines 23-25 illustrate the cases component-to-component, component-to-port, and port-to-component.

Component-to-Component Line 23 specifies that component `SimulationPanel` and component `Environment` are connected. In C&C models components can only be connected via corresponding ports — these are not required to be given for abstract connectors in views. Required ports, e.g., an incoming port of component `Environment` can be omitted in C&C views as long as their name is not referenced in the definition of a connector.

Component-to-Port Line 24 specifies a connection between component `UserButtonReader` and component `SimulationPanel`. The endpoint of the connector is the port `button` of component `SimulationPanel`. Similarly, MontiArcViews allows the definition of Port-to-Component connectors (line 25) and Port-to-Port connectors (see Definition 3.6, Item 3 for the types of abstract connectors in C&C views).

Connectors Crossing Components Line 25 specifies a connection between component `PreProcessor`'s port `trans` and component `Environment`. In this view the connector crosses the border of component `SimulationPanel` that contains component `Environment`. Similarly, if the components `UserButtonReader` and `SimulationPanel` are not immediate siblings in a satisfying C&C model the connector in line 24 might cross other component's interfaces requiring corresponding ports and further connectors not shown in the view.

Abstract connectors in views specify a chain of connectors in the C&C model — the data is passed as is: no intermediate processing by components is allowed unless these only forward the data by (chains of) connectors.

3.6.5. Extension Points for Additional Language Features

The C&C model language variability mechanism described in Definition 3.6 offers extension points for additional language features by attaching stereotypes to components, ports, and connectors. This variability mechanism directly maps to stereotypes in the MontiArc ADL. In the concrete syntax of MontiArc, stereotypes are enclosed by double angle brackets («...»). Multiple stereotypes are placed as a comma separated list — within opening and closing angle brackets — before the element they apply to (e.g., «`atomic,interfaceComplete`» in front of a component).

Atomic components The stereotype «`atomic`» in front of the component `Sensor` (Listing 3.15, l. 6) requires that the component is atomic in any satisfying C&C model. The component must not have subcomponents in the view (see Section 3.3.1).

Interface-complete components The stereotype «`interfaceComplete`» in front of the component `SimulationPanel` (Listing 3.15, l. 9) means that its interface is completely specified in the model and thus consists of the single outgoing port `button` of type `Button`. When using the stereotype «`interfaceComplete`» all ports in the interface of the component have to be listed at least with a name and direction. Together with the stereotype «`interfaceComplete`» it is possible to use the stereotype «`untyped`» but not the stereotype «`unnamed`» for individual ports of the marked component (see Section 3.3.1).

Further extensions of C&C views with stereotypes require adapting the well-formedness rules for C&C views as shown for the two above examples in Section 3.3.1 and adapting

MontiArcView

```
1  package pumpStationExample;
2
3  <<view>> component UserButtonExtended {
4
5    component UserButtonReader {
6      <<atomic>> component Sensor { }
7    }
8
9    <<interfaceComplete>> component SimulationPanel {
10     port
11       out Button button;
12   }
13 }
```

Listing 3.15: A view of the system called `UserButtonExtended` with stereotypes «`atomic`» and «`interfaceComplete`».

the satisfaction relation between C&C models and C&C views as shown for the two examples in Section 3.4.1.

3.7. Discussion and Views Related Concepts

In this section we discuss some of the design decisions of C&C views, we discuss the usage of the term view in the field of software architecture, and we discuss concepts of ADLs that are related to our notion of C&C views .

3.7.1. Corresponding Elements are not Necessarily Independent

According to the definition of satisfaction of a C&C view by a C&C model (see Definition 3.8) two different elements shown in a C&C view may have the same corresponding element in a C&C model. Thus, corresponding elements are not necessarily independent.

The satisfaction of a C&C view by a C&C model in Definition 3.8 is defined based on the satisfaction of the view's elements, e.g., the satisfaction of an abstract connector by a concrete chain of connectors in the C&C model (Definition 3.8, Item 5). The definition does not require independence of the C&C model elements satisfying separate elements of the C&C view. We give two examples for the view `UserButtonEnvPS` shown in Figure 3.16.

The abstract connector from the port `button` of the component `SimulationPanel` to the component `UserButtonReader`, the abstract connector from component `Environment` to component `UserButtonReader`, and the abstract connector from component `Environment` to component `PumpingSystem` are all three satisfied by the chain of connectors shown in the witness for the view `UserButton` shown in Figure 3.5 although they appear as separate elements in the view. In fact, the chain of connectors

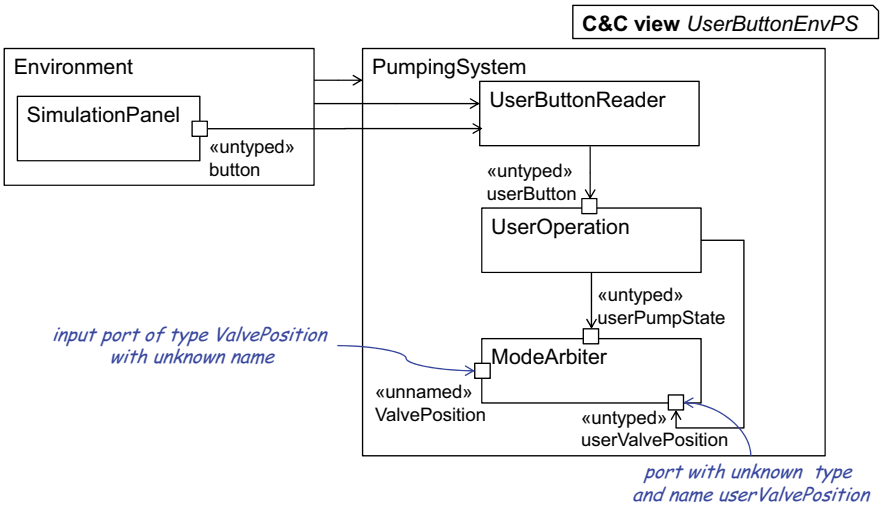


Figure 3.16.: A modified version of the C&C view `UserButtonFlow` shown in Figure 3.3. The unnamed input port of type `ValvePosition` and the port `userValvePosition` with unknown type are satisfied by a the single port `userValvePosition` of type `ValvePosition` in the C&C model `PumpStation`.

shown in the view is the only way to satisfy the first two abstract connectors in the C&C model `PumpStation` since the component `UserOperation` only has a single incoming connector. The top-most of the three abstract connectors has multiple corresponding chains of connectors in the C&C model `PumpStation`.

As a second example, the unnamed port of the type `ValvePosition` of component `ModeArbiter` and the port `userValvePosition` of the same component with an unknown type are both satisfied by the single port `userValvePosition` of the type `ValvePosition` in the C&C model `PumpStation`. An alternative match for the unnamed port in the C&C model is the port `emsValvePosition` of the same type.

If the independence of matching elements in the C&C model was required, the C&C model `PumpStation` would not satisfy the C&C view `UserButtonEnvPS`. There would be no independent chains of connectors for the two abstract connectors to component `UserButtonReader`. We consider a user study of C&C views satisfaction definitions with different treatments of the independence of elements an interesting future work.

3.7.2. Unnamed Components in Component and Connector Views

In our definition of C&C views we decided to require a name for every component in the view while, e.g., ports might be unnamed. We believe that it would be possible to also

allow unnamed components and define the correspondence of the unnamed component based on the hierarchy and connectedness information given for the unnamed component in the C&C view. In this case an unnamed component might have multiple possible matches in the C&C model and in case of non-satisfaction of the view the reasons for non-satisfaction would depend on the different matches of the corresponding component.

We believe that for specifying and documenting structural properties of C&C models and for understanding verification results components that always have a name are more intuitive. Nevertheless, we consider unnamed components an interesting extension of the basic C&C views language and its expressiveness.

3.7.3. Component and Connector Views for Component Type Definitions

We have formally defined the satisfaction relation between C&C models and C&C views in Section 3.4. One of the main purposes of C&C views is the specification of relations between components that are identified by their names in a C&C model. Many ADLs allow modeling of C&C models in three steps by first defining component types, secondly defining component implementations, and finally instantiating component implementations (note that MontiArc [HRR12] combines the first and second step for composed components).

One might want to extend the satisfaction relation between a C&C model and a C&C view to a set of component type definitions and a C&C view. Component type definitions introduce names for their subcomponents. These names are not necessarily unique for all component type definitions and component types may be instantiated multiple times. We have illustrated the relation between component type definitions and C&C models in Section 6.2.1, Figure 2.5. The component names in the C&C model resulting from the component type definition instantiation are chosen as full qualified names derived from the subcomponent hierarchy.

An extension of C&C views to describe the instantiation of component type definitions would require establishing a name mapping from components in C&C views to components instantiated from component type definitions. Furthermore, to trace a property formulated in a C&C view might require the inclusion of component type names in C&C views. We consider investigating possible extensions of C&C views and the satisfaction relation to handling component type definitions a future work.

3.7.4. Component and Connector Views to Model Variability

We have introduced various usage scenarios for C&C views in Section 3.2.1. The documentation usage scenario presented is based on works by Grönniger et al. [GHK+07, GHK+08b] about feature modeling. C&C views are applied to modeling feature views on a given C&C model expressed in SysML [Obj12b] that only contain model elements of interest for a given feature in an automotive function network.

The same group of authors further investigated the application of C&C views for variability modeling for C&C models. To model software product variability C&C models are used as 150% models — containing many variants of one logical architecture in one

model — and every view describes a concrete product [GKPR08, GHK+08a]. Modeling 150% models requires different well-formedness rules for C&C models, e.g., a port might be the target of multiple connectors as long as at most one of the incoming connectors is selected in every valid configuration.

3.7.5. Component and Connector Views Related Concepts

Our notion and terminology of C&C views is based on works by Grönniger et al. [GHK+07, GHK+08a, GHK+08b, GKPR08]. The term view is however very general and used in the modeling and software architecture literature in various ways.

The term architectural view

Software architecture deals with high-level descriptions of a software system structure and behavior [GS94]. In the software architecture literature, the terms *view* and *architectural view* are used in a rather broad and sometimes informal way.

Kruchten [Kru95] divides the documentation of a software architecture in 4+1 "concurrent views, each one addressing one specific set of concerns". The views identified are the logical view, the process view, the physical view, the development view, and scenarios combining these views. This usage of the term view is on a different conceptual level than in our work. Kruchten uses notations similar to C&C models in the process view and the development view [Kru95].

Many very general definitions of views exist in the software architecture literature. For example, Taylor et al. [TMD09] define a viewpoint as "a perspective from which a view is taken" and a view as "a set of design decisions related by a common concern (or set of concerns)". In the IEEE standard 1471 [IEE], a view is defined as "a representation of a whole system from the perspective of a related set of concerns". Giese and Vilbig [GV06] define that an architectural view of a system "represents a partial software C&C model dedicated to a particular concern". Finally, the book by Clements et al. [CBB+10] defines a view as a "representation of a set of system elements and the relationships associated with them". Others have very concrete notions of views, e.g., Sabetzadeh and Easterbrook [SE04, SE06] refer to views as parts of partial models that they represent as typed graphs.

We focus on a structural viewpoint of software architecture. Our notion of view for the structure of component and connector C&C models falls within the broad definitions cited above.

Related specification mechanisms in ADLs and modeling languages

We have presented C&C views for the specification and documentation of crosscutting concerns for C&C models and formally defined the satisfaction relation between C&C models and C&C views.

The satisfaction relation between a C&C model and a C&C view might be seen as a refinement relation where the C&C model refines the C&C view. Many works in software

architecture deal with refinement relations between architectural elements. Broy and Stølen [Bro93, BS01] introduce glass-box refinement where a refinement has to respect the decomposition of a specification into components and directed channels between them. Philipps and Rumpe [PR97, PR99] have developed a set of proven refinement rules based on [Bro93]. Moriconi et al. [MQR95] have investigated a notion of what they call refinement between architectural models. However, the notion of refinement from [Bro93, PR97, PR99, BS01] and ours are fundamentally different from [MQR95] as the C&C model may add elements (components, ports, connectors) that do not appear in the view. To the best of our knowledge the AADL [FG12] is the only ADL that directly supports refinement statements for architectural elements. A distinction from our work common to all refinement approaches above is that these approaches allow to extend but require to preserve component hierarchies and thus do not offer features as abstraction over component hierarchy and abstract connectors.

Chechik et al. [FBDCS11, SFC12] introduce a language independent mechanism for incomplete models. Their approach operates on the syntax definition of a modeling language and adds annotations to mark an element as optional (*May*), as abstract representing sets of concrete elements (*Abs*), as possibly identical with another element (*Var*), and to mark the whole model as incomplete (*OW*). The incomplete models developed by Chechik et al. may appear similar to C&C views treated as incomplete C&C models because elements in C&C views can by default be considered abstract (*Abs*) and incomplete (*OW*). The approach of Chechik et al. is however generic and language independent while C&C views offer domain-specific abstraction mechanism and a corresponding satisfaction relation, e.g., abstract connectors can not easily be expressed in a generic approach. Abstract connectors require weakening specific well-formedness rules of the original language, e.g., allow crossing of component boundaries.

C&C views are a specification mechanism for structural models. It is important not to confuse behavioral views and structural ones. There are many behavioral specification languages and many of them have related views. For example, linear temporal logic (LTL) [MP92, Pnu77] is a behavioral specification language, and scenarios, expressed, e.g., using live sequence charts (LSC) [DH01, HM08], may be considered as related behavioral views. In contrast, C&C views are intentionally limited to structural properties, expressed using structural views. In the behavioral case, a system behavior is typically described using a state machine. The properties it needs to satisfy are expressed using LTL formulas or scenarios.

Chapter 4.

Component and Connector Views Verification

A C&C view documents design decisions and relations between the elements of a C&C model. It is thus a specification that a C&C model should satisfy. We have formally defined the satisfaction relation between C&C models and C&C views in Section 3.4.

When C&C views are created as specifications a C&C model should satisfy or when they are employed as documentation to highlight design decisions, it is useful to automatically determine whether a C&C model satisfies a C&C view. In this chapter we present an algorithm for the verification of C&C models with respect to C&C views.

The problem we solve is as follows: Given a C&C model m and a C&C view $view$, decide whether $m \vDash view$. Moreover, in addition to the Boolean answer, we are interested in generating informative formal witnesses in the form of views, each of which will serve as a concrete proof for satisfaction or non-satisfaction and help the engineer understand the satisfaction checking result and the result's root causes.

A witness justifies a positive verification result by listing the elements of the C&C model implementing the C&C view or it justifies a negative verification result by exhibiting a small subset of the C&C model that violates the view.

We have reported on parts of this work in [MRR14].

Chapter outline and contributions

The next section refers to the C&C model of the pump station example introduced in Section 3.1 and adds C&C views for C&C views verification. We formally introduce the C&C views verification problem in Section 4.2. Section 4.3 presents our algorithms for checking satisfaction and generating witnesses. The polynomial algorithms for solving the C&C views verification problem and generating witnesses justifying verification results are a major contribution of this thesis.

Section 4.4 describes our prototype tool implementation and the evaluation we have conducted. Section 4.5 discusses advanced topics related to C&C views verification, strengths, and limitations. We review related work in Section 4.6.

4.1. Component and Connector Views Verification Example

Figure 4.1 shows the complete C&C model of the pump station system adopted as an example from the AutoFOCUS IDE [HF07]. It consists of 16 components, a containment hierarchy of four levels, and 47 connectors. The complete C&C model with all port names and types is presented in Appendix G. The C&C model is also available in textual MontiArc format with supporting materials from [wwwu]. Please note the difference in the representations of the same C&C model in Figure 3.2 and Figure 4.1: the `PumpStation` system is shown in five compartments in Figure 3.2, one for each component and its immediate subcomponents. In Figure 4.1 we show the C&C model in its complete depth with four levels of hierarchy to give a comprehensive perspective of the C&C model.

Figure 4.2 (a) shows a C&C view named `ASPumpingSystem`. This view focuses only on the connections between sensors and actuators in the system. As `ASPumpingSystem` is a view, it does not contain all components and connectors. While the components shown inside the `PumpingSystem` component must actually be inside it, they may be nested within some of its subcomponents (not shown in this view). Finally, each of the sensors shown must be connected to the corresponding actuator in the C&C model, but the connection between them is not necessarily direct. It is easy to see that the C&C model satisfies the view: all components mentioned in the view are present in the C&C model, the C&C model respects the containment relation specified by the view, and all connectors in the view have corresponding chains of connectors in the C&C model. Satisfaction is denoted $\texttt{PumpStation} \models \texttt{ASPumpingSystem}$. We have given the formal definition of when does a C&C model satisfy a view in Section 3.4.

Figure 4.2 (b) shows another C&C view, named `UserButton`. The view focuses on the components and their connectors that participate in a specific use case, namely, when the user presses the button. Again, not all components are shown, but only the ones participating in this use case. Please note that the containment hierarchy is not shown, and the connectors are abstract, i.e., they specify connections, but not necessarily direct ones. It is easy to see that the C&C model satisfies this view: all the components mentioned in the view exist in the C&C model, and all abstract connectors in the view have corresponding chains of concrete connectors.

Figure 4.3 (a) shows a C&C view named `PCPumpingSystem`, which includes a connection from the `PhysicsSimulation` component to the `Controller` component, both within the `PumpingSystem` component. It is easy to see that the C&C model does not satisfy the view. We denote this by $\texttt{PumpStation} \not\models \texttt{PCPumpingSystem}$. First, in the view `PCPumpingSystem`, component `PhysicsSimulation` is inside component `PumpingSystem`, while in the C&C model it is not. Second, the connector from `PhysicsSimulation` to `Controller` shown in the view does not have a corresponding connector or chain of connectors in the C&C model.

Finally, Figure 4.3 (b) shows the C&C view `SystemEmergencyController`, which specifies structural properties related to a use case of emergency. Again, the pump station C&C model does not satisfy this view. First, the view includes the `Emergency-`

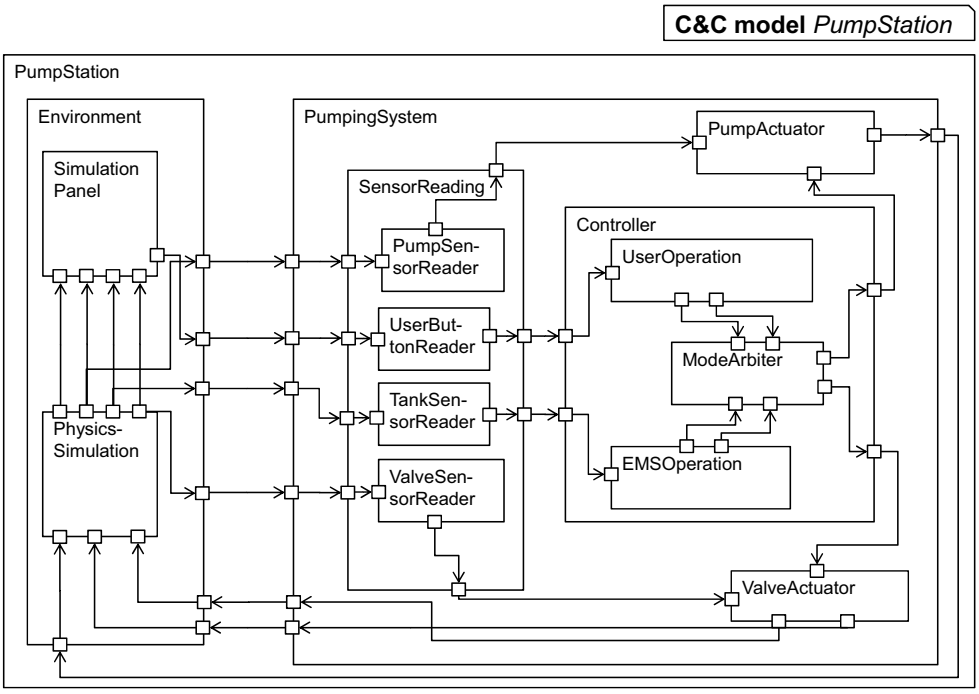


Figure 4.1.: The C&C model of the pump station. Here we show the C&C model with its complete depth in one figure, in order to give a comprehensive perspective. To avoid clutter we omit port names and types from the figure. However, as this is a C&C model (and not a view), all ports have names and types. For example, the type of the upper left incoming port of the component `ModeArbiter`, with a connector coming from the component `UserOperation`, is `Boolean` and its name is `userPumpState`. The complete model with all port names and types is shown in Appendix G.

`Controller` component, which does not exist in the C&C model. Second, the type `Integer` of the port named `userPumpState` of the `ModeArbiter` component, does not match the type `Boolean` of the port with the same name `userPumpState` in the C&C model.

Please note that the views shown above crosscut the traditional boundaries of systems and subsystems. They abstract the hierarchy (or parts of the hierarchy) away and instead focus on the components and connectors participating in a use case, e.g., the user pressing a button, or a certain concern, e.g., an emergency system. We consider this to be an important feature of our work.

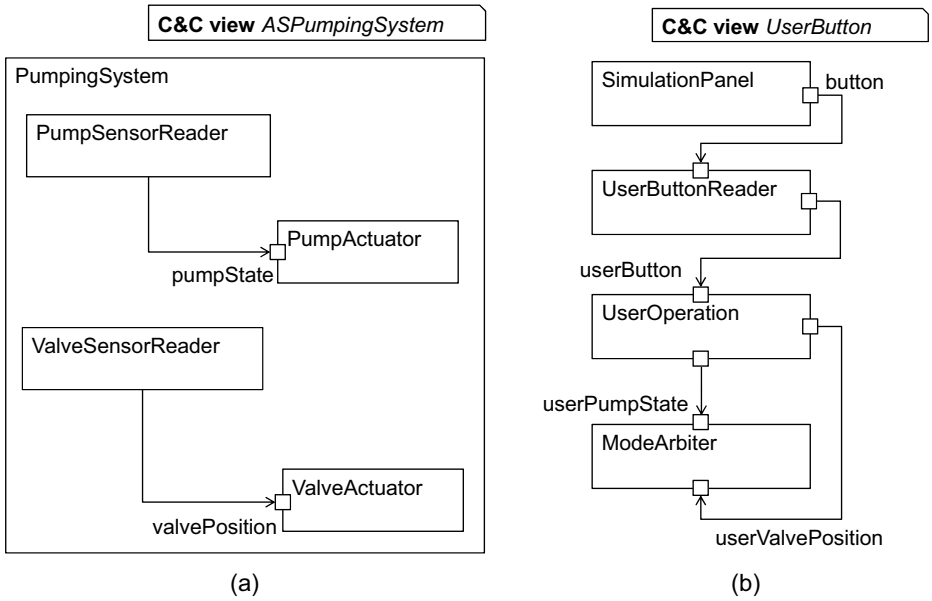


Figure 4.2.: Two C&C views: `ASPumpingSystem` (a) and `UserButton` (b). Please note that as these are views, they allow one not to fully specify ports, port types, and port names. For example, the abstract connector going out from `PumpSensorReader` in the `ASPumpingSystem` view has no specified source port. The C&C model `PumpStation` satisfies these views.

4.1.1. Witnesses for Satisfaction and Non-Satisfaction

As mentioned above, given a C&C model and a view, we are interested in checking whether the former satisfies the latter. However, in practice, a Boolean answer is by far not enough. Rather, we are interested in concrete proofs for satisfaction or non-satisfaction. These should enable comprehension and in the case of non-satisfaction, point the architect to the problems and assist her in correcting her design.

As an example, Figure 4.4 shows a view which serves as a witness for showing that `PumpStation` $\vDash$ `UserButton`, that is, that the C&C model `PumpStation` satisfies the view `UserButton`. Please note the complete hierarchy (excluding siblings) up until the least common parents of the components appearing in the view, and the chain of concrete connectors corresponding to the abstract connectors in the view, i.e., the chain from `SimluationPanel` to `ModeArbiter`. This information demonstrates satisfaction by showing the elements of the C&C model required to satisfy the view and their hierarchical containment in the relevant subsystem.

As another example, Figure 4.5 shows two views which serve as witnesses for showing that `PumpStation` $\nvDash$ `PCPumpingSystem`, that is, that the C&C model `PumpStation`

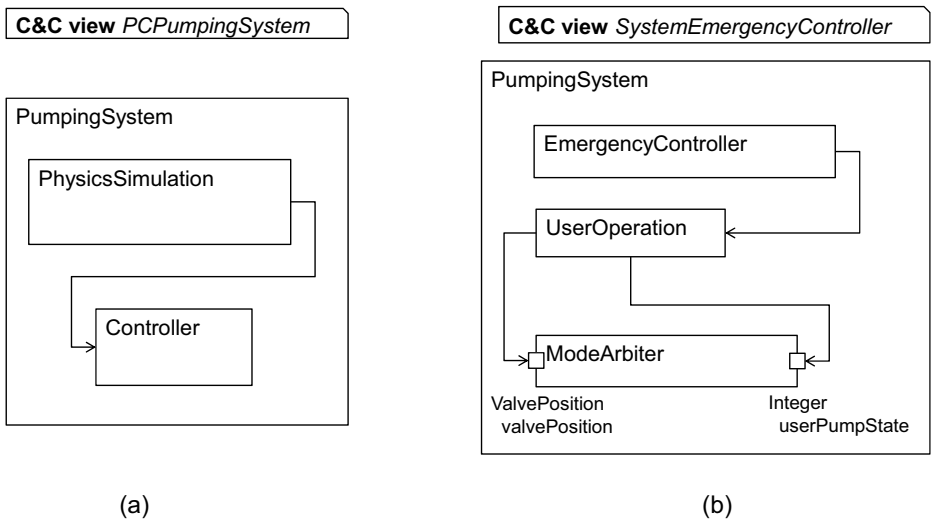


Figure 4.3.: Two C&C views: `PCPumpingSystem` and `SystemEmergencyController`. The C&C model `PumpStation` does not satisfy these views.

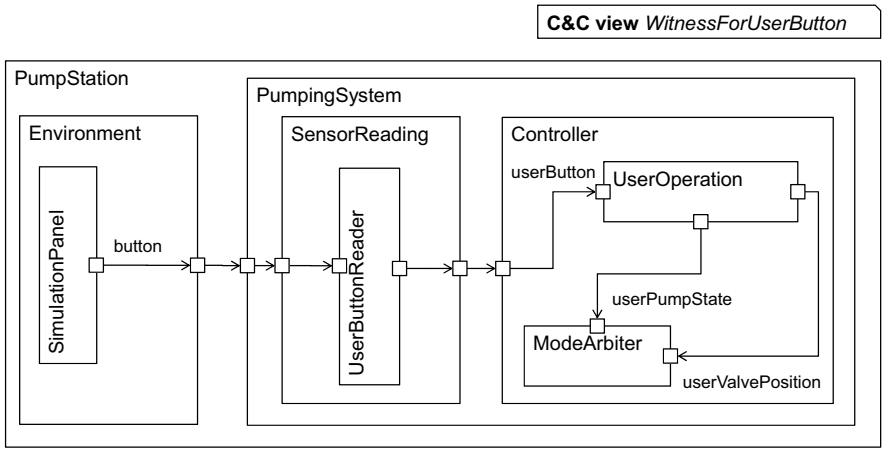


Figure 4.4.: Generated witness for satisfaction of the `UserButton` view.

does not satisfy the view `PCPumpingSystem`. While in the view `PCPumpingSystem` the `PumpingSystem` component contains the `PhysicsSimulation` component, in the C&C model, as shown in the witness in Figure 4.5 (a), they are independent. While in `PCPumpingSystem` there is a connection from an unnamed port of component `PhysicsSimulation` to an unnamed port of component `Controller`, in the

C&C model, as shown in the witness in Figure 4.5 (b), the component `Controller` is not in the set of components reachable with a chain of connectors from component `PhysicsSimulation`. Please note that the two witnesses include annotations which explain, in natural language, the relevant reason for non-satisfaction.

In Section 4.3 we show the algorithms to decide satisfaction and to generate informative witnesses like the ones we show above.

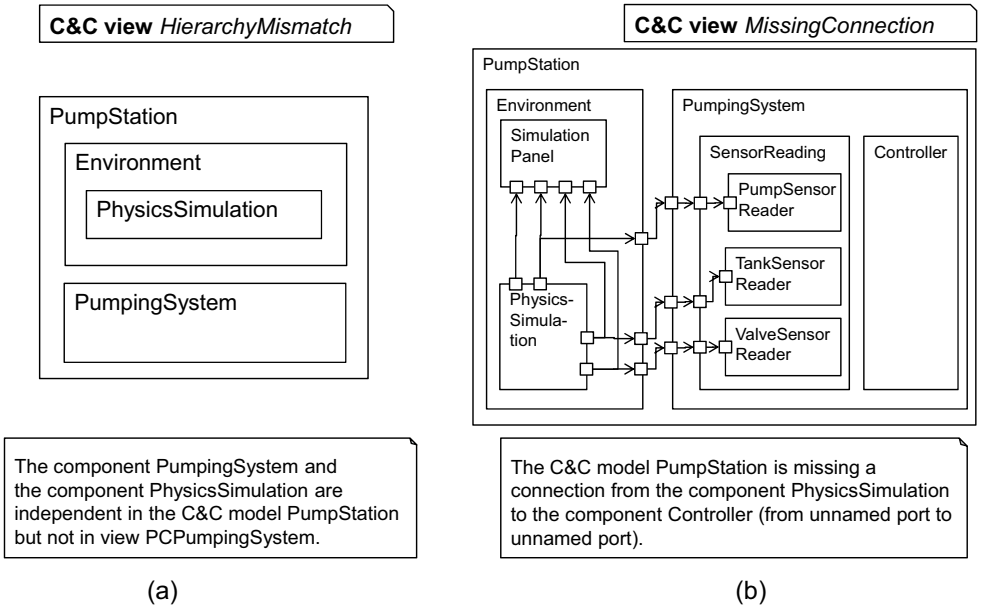


Figure 4.5.: Two generated witnesses for non-satisfaction of the `PCPumpingSystem` view.

4.2. Component and Connector Views Verification Problem

The C&C views verification problem for a C&C view and a C&C model is defined in Definition 4.6. The input for the verification problem is a C&C view (defined in Definition 3.6) and a C&C model (defined in Definition 2.2). The output of the verification problem is a Boolean answer whether the C&C model satisfies the C&C view.

Definition 4.6 (C&C views verification problem)**.** Given a C&C view *view* and a C&C model m determine whether $m \vDash view$. △

In addition to a Boolean answer to the verification problem, we are interested in constructing witnesses that demonstrate the verification result. In case the C&C model

satisfies the view, the witness should be a minimal subset of the C&C model that is by itself a well-formed C&C view and contains components (including their concrete hierarchical relations), ports, and connectors of the C&C model that satisfy all elements appearing in the view.

In case the C&C model does not satisfy the view, we are interested in a set of witnesses, each of which should explain one reason for non-satisfaction. We list all classes of reasons for non-satisfaction and the elements provided in each witness in Section 4.3.2. The elements contained in the generated witnesses for non-satisfaction are implementation-specific, while the classification of the reasons for non-satisfaction is independent of a concrete implementation and directly follows from the view's semantics defined in Definition 3.8.

4.3. Checking Satisfaction and Generating Witnesses

Our algorithm checks whether a given C&C model satisfies a given C&C view. In case the C&C model satisfies the view our algorithm computes a witness for satisfaction including all elements shown in the view and their concrete relations in the C&C model to support engineers in finding view's elements in the architecture. In case the C&C model does not satisfy the view the algorithm computes witnesses for non-satisfaction. Each witness for non-satisfaction shows the relevant subsystem and the elements of the C&C model that violates the C&C view and thus may help an engineer understand the exact reasons for non-satisfaction.

As witnesses are meant for human comprehension, our algorithm generates minimal witnesses for non-satisfaction with respect to the elements required by the witnesses for different reasons for non-satisfaction. In case of satisfaction the witness is not necessarily unique. For satisfaction the algorithm employs a heuristics to generates a small but not necessarily minimal witness (see the description of the algorithm in Section 4.3.3 and the discussion of minimality in Section 4.5.2).

We start with a description of the different kinds of witnesses we are interested in and then give an overview of the algorithm. We prove the correctness and completeness of the algorithm and discuss its complexity.

4.3.1. A Witness for Satisfaction

In case the C&C model satisfies the view, the algorithm outputs a positive answer and produces a witness: a minimal subset of the C&C model that contains all the components appearing in the view and their parent components up until their least common parent, all the C&C model ports corresponding to the ports appearing on all components in the view, and chains of C&C model connectors representing all abstract connectors appearing in the view.

For example, recall the two views shown in Figure 4.2. Both views are satisfied by the pump station C&C model. Figure 4.4 shows the generated witness for satisfaction of the `UserButton` view (see Section 4.1), and Figure 4.7 shows the generated witness

for satisfaction of the view `ASPumpingSystem`. Please note the complete hierarchy (excluding siblings) up until the least common parents of the components appearing in the views, e.g., for the view `ASPumpingSystem`, the least common parent component is the component `PumpingSystem` as shown in the witness in Figure 4.7. Also note the chain of concrete connectors corresponding to the abstract connectors in the views, e.g., the chain from `PumpSensorReader` to `PumpActuator`.

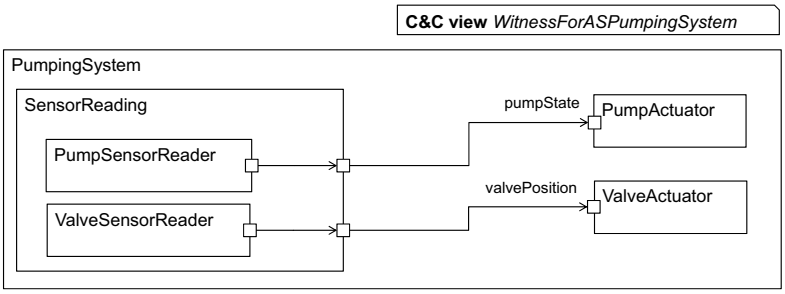


Figure 4.7.: Generated witness for satisfaction of the `ASPumpingSystem` view.

4.3.2. Witnesses for Non-Satisfaction

In case the C&C model does not satisfy the view, our algorithm outputs a negative answer and produces a set of witnesses. The witnesses are divided into four classes according to four reasons of non-satisfaction as follows:

- **Missing Component**: the view contains a component that does not exist in the C&C model (MISS_COMP) — the generated witness, one for each missing component, is an empty view annotated with the name of the missing component;
- **Hierarchy Mismatch**: the view contains two components that are in a different containment relation in the C&C model, more specifically
  - HIER_IND_IN_VIEW: independent in the view but not independent in the C&C model
  - HIER_IND_IN_CNCM: not independent in the view but independent in the C&C model
  - HIER_REV_CONT: reverse containment relation in the C&C model

  the generated witness, one for each hierarchy mismatch, consists of the relevant pair of components up to their least common parent in the C&C model but without siblings and without any connectors and ports;
- **Interface Mismatch**: the view contains a component with a port that does not exist in the C&C model, more specifically

  - PORT_WRONG_TYPE: the port has a wrong type in the C&C model
  - PORT_WRONG_DIR: the port has the wrong direction in the C&C model
  - PORT_NO_MATCH: no port in the C&C model matches the port

  the generated witness, one for each interface mismatch, consists of the relevant component in the C&C model without subcomponents with the relevant port or its complete interface, for the non-satisfaction reason PORT_NO_MATCH, annotated with the port from the view that has no match; and

- **Missing Connection**: the view contains an abstract connector that has no corresponding concrete chain of connectors in the C&C model (CONN_NO_MATCH) — the generated witnesses consist of the relevant pairs of components up to their least common parent in the C&C model but without siblings and with all components reachable by connectors from the source component.

In Section 4.3.5 we show that this classification of four reasons for non-satisfaction is complete, that is, given a C&C model m and a view *view*, if $m \not\models view$ then at least one of the reasons above holds.

Moreover, note that each of the witnesses for non-satisfaction as well as for satisfaction is by itself a view that is satisfied by the C&C model. The witness is a C&C view with concrete containment and abstract connectors each matching exactly one connector in the C&C model. The satisfaction of the witness by the C&C model can be checked by the same algorithm. This has two advantages. First, the engineer does not need to learn a new language in order to understand the witness. Second, the same tools applied to the views, e.g., for presentation or further analysis, may be applied to the witnesses.

All witnesses have some common properties. First, each contains a single least common parent component, which is the top of the relevant subsystem (except for the case of a missing component, see the discussion in Section 4.5). Second, each is a view with concrete containment and connectors. Finally, each witness includes complete port information of name, type, and direction (although the set of ports shown is not necessarily complete). Interestingly, all witness are their own witness for satisfaction, when checked against the same C&C model.

Example witnesses for non-satisfaction

For example, as presented in Section 4.1, for the pump station C&C model and the view `PCPumpingSystem` (see Figure 4.1 and Figure 4.3 (a)), our algorithm provides a negative answer and the two witnesses shown in Figure 4.5.

First, the algorithm finds a hierarchy mismatch — in the view the `PumpingSystem` component contains the `PhysicsSimulation` component but in the C&C model they are independent — and generates the view shown in Figure 4.5 (a), asserting that `PumpingSystem` and `PhysicsSimulation` are independent. Second, the algorithm finds a missing connection — the C&C model is missing a connection from component `PhysicsSimulation` to component `Controller` — and generates the view shown in

Figure 4.5 (b), asserting that component `Controller` is not in the set of components reachable with a chain of connectors from component `PhysicsSimulation`.

As another example, when checking whether the C&C model `PumpStation` shown in Figure 4.1 satisfies the view `SystemEmergencyController` shown in Figure 4.3 (b), our algorithm provides a negative answer and produces three witnesses.

First, the algorithm finds two interface mismatches — the type of port `userPumpState` of component `ModeArbiter` is `Integer` in the view but `Boolean` in the C&C model, and there is no match for the port `valvePosition` of component `ModeArbiter` in the C&C model. For these, the algorithm generates the witnesses shown in Figure 4.8. Second, the algorithm finds a missing component — the component `EmergencyController` is not present in the C&C model — and generates an empty view as a witness for non-satisfaction.

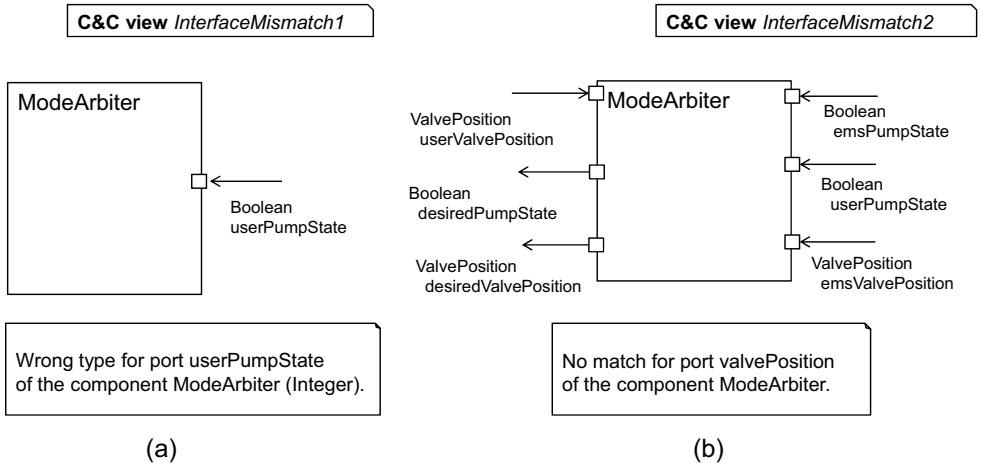


Figure 4.8.: Generated non-satisfaction witnesses for the view `SystemEmergencyController`.

Natural language descriptions

Finally, for each of the generated witnesses, in each of the four classes, we generate a detailed description in natural language.

For example, for the hierarchy mismatch between the pump station C&C model and the view `PCPumpingSystem`, as discussed above, we generate the following text:

> `The component PumpingSystem and the component PhysicsSimulation are independent in the C&C model PumpStation but not in the view PCPumpingSystem.`

These generated texts appear on the witnesses view in the Eclipse plug-in (see Section 4.4) and as a comment in the generated witness's document (see, e.g., Figure 4.8 (a) and (b)). The generated text is intended to further help the engineer identify the cause for non-satisfaction.

The natural language descriptions are generated using templates as we describe in Section 4.3.4.

4.3.3. Algorithms Overview

The input for the algorithm consists of a C&C model and a view. The output is a Boolean answer, whether the C&C model satisfies the view or not, and a set of one or more witnesses. The algorithm works by checking for reasons of non-satisfaction of the view and the C&C model in the four classes described in Section 4.3.2. The checks are done independently and the answer that the C&C model satisfies the view is returned if and only if no reason for non-satisfaction is found.

We have implemented the algorithms as an Eclipse plug-in written in Java. The implementation is based on the MontiCore framework as described in Section 4.4.

Procedure 1 contains a pseudo code for the algorithm computing the non-satisfaction reasons for a C&C model m and a C&C view *view*. The algorithm works by checking for non-satisfaction reasons of each class separately: in Procedure 2 for the non-satisfaction reason Missing Component, in Procedure 3 for the non-satisfaction reason Hierarchy Mismatch, in Procedure 4 for the non-satisfaction reason Interface Mismatch, and in Procedure 5 for the non-satisfaction reason Missing Connection.

Procedure 1 *computeNonSatReasons*$(m, view)$

```
1: define NonSatReasons as set of NonSatisfactionReason
2: add all computeNonSatReasonsMissingComponent(m, view) to NonSatReasons
3: add all computeNonSatReasonsHierarchyMismatch(m, view) to NonSatReasons
4: add all computeNonSatReasonsInterfaceMismatch(m, view) to NonSatReasons
5: add all computeNonSatReasonsMissingConnection(m, view) to NonSatReasons
6: return NonSatReasons
```

Procedure 2 *computeNonSatReasonsMissingComponent*$(m, view)$

```
1: define NonSatReasons as set of NonSatisfactionReason
2: for all missingCmp ∈ (view.Cmps ∖ m.Cmps) do
3:    add (MISS_COMP, missingCmp) to NonSatReasons
4: end for
5: return NonSatReasons
```

The algorithm's pseudo code uses the structures and functions of the definitions of C&C models from Definition 2.2 and C&C views from Definition 3.6. In our implementation — based on MontiArc — the operators for membership of components, and types are defined by name. The names of components are unambiguous in every C&C model and view. The comparison of ports is defined in accordance to Definition 3.8, Item 4 (b)

Procedure 3 *computeNonSatReasonsHierarchyMismatch*$(m, view)$

```
1: define NonSatReasons as set of NonSatisfactionReason
2: for all cmp ∈ (view.Cmps ∩ m.Cmps) do
3:    for all subCmp ∈ (view.subs(cmp) ∩ m.Cmps) do
4:       if cmp is not parent of subCmp in m then
5:          if subCmp is parent of cmp in m then
6:             add (HIER_REV_CONT, cmp, subCmp) to NonSatReasons
7:          else
8:             add (HIER_IND_IN_CNCM, cmp, subCmp) to NonSatReasons
9:          end if
10:      end if
11:   end for
12:   for all cmp′ ∈ (view.Cmps ∩ m.Cmps ∖ cmp) do
13:      if cmp is not parent of cmp′ in view ∧ cmp′ is not parent of cmp in view then
14:         if cmp is parent of cmp′ in m then
15:            add (HIER_IND_IN_VIEW, cmp, cmp′) to NonSatReasons
16:         else if cmp′ is parent of cmp in m then
17:            add (HIER_IND_IN_VIEW, cmp′, cmp) to NonSatReasons
18:         end if
19:      end if
20:   end for
21: end for
22: return NonSatReasons
```

Procedure 4 *computeNonSatReasonsInterfaceMismatch*$(m, view)$

```
1: define NonSatReasons as set of NonSatisfactionReason
2: for all cmp ∈ (view.Cmps ∩ m.Cmps) do
3:    for all p ∈ view.ports(cmp) do
4:       if view.name(p) = ⊥ then
5:          if ∄p′ ∈ m.ports(cmp) : view.dir(p) = m.dir(p′) ∧ view.type(p) ∈ {m.type(p′), ⊥}
          then
6:             add (PORT_NO_MATCH, cmp, p) to NonSatReasons
7:          end if
8:       else if ∀p′ ∈ m.ports(cmp) : m.name(p′) ≠ view.name(p) then
9:          add (PORT_NO_MATCH, cmp, p) to NonSatReasons
10:      else
11:         define p′ as THE p′ ∈ m.ports(cmp) : m.name(p′) = view.name(p)
12:         if view.dir(p) ≠ m.dir(p′) then
13:            add (PORT_WRONG_DIR, cmp, p) to NonSatReasons
14:         end if
15:         if view.type(p) ≠ ⊥ ∧ view.type(p) ≠ m.type(p′) then
16:            add (PORT_WRONG_TYPE, cmp, p) to NonSatReasons
17:         end if
18:      end if
19:   end for
20: end for
21: return NonSatReasons
```

Procedure 5 *computeNonSatReasonsMissingConnection*$(m, view)$

```
 1: define NonSatReasons as set of NonSatisfactionReason
 2: for all ac ∈ view.AbsCons with ac.srcCmp, ac.tgtCmp ∈ m.Cmps do
 3:    if ac is cmp-to-cmp ∧ ∄p, p′ with (ac.srcCmp, p) →* (ac.tgtCmp, p′) in m then
 4:       add (CONN_NO_MATCH, ac) to NonSatReasons
 5:    else if ac is cmp-to-port ∧ ∄p, p′ with ac.tgtPort ≅ p′∧
                                    (ac.srcCmp, p) →* (ac.tgtCmp, p′) in m then
 6:       add (CONN_NO_MATCH, ac) to NonSatReasons
 7:    else if ac is port-to-cmp ∧ ∄p, p′ with ac.srcPort ≅ p∧
                                    (ac.srcCmp, p) →* (ac.tgtCmp, p′) in m then
 8:       add (CONN_NO_MATCH, ac) to NonSatReasons
 9:    else if ac is port-to-port ∧ ∄p, p′ with ac.srcPort ≅ p ∧ ac.tgtPort ≅ p′∧
                                    (ac.srcCmp, p) →* (ac.tgtCmp, p′) in m then
10:       add (CONN_NO_MATCH, ac) to NonSatReasons
11:    end if
12: end for
13: return NonSatReasons
```

based on the containing component and port name. In case the name of a port p is omitted in the view the implementation iterates over the ports p' of the corresponding component in the C&C model and checks for a match between ports p and p' by checking $view.type(p) \in \{m.type(p'), \bot\} \wedge view.dir(p) = m.dir(p')$. This check is required, e.g., in Procedure 4, l. 5 for the non-satisfaction reason Interface Mismatch.

The algorithm's pseudo code uses high-level operations, e.g., for two components c_1 and c_2 the check c_1 **`is parent of`** c_2 **`in`** m or *view* checks that the component c_1 is a not necessarily direct parent component of the component c_2 in the C&C model m or the C&C view *view*.

We formally state and prove the correctness and completeness of the algorithms in Section 4.3.5. We discuss the time complexity of the algorithms and the maximal numbers of witnesses computed in Section 4.3.6.

Algorithm for generating witnesses for satisfaction

If no non-satisfaction reasons are produced by any of the checks, the C&C model satisfies the view and the algorithm generates a positive witness as shown in Procedure 6.

Witness generation starts with an empty view *witness* (l. 1). The witness is populated with elements from the C&C model m that witness the view *view*'s components and hierarchy (ll. 5-9), abstract connectors (ll. 10-15), ports (ll. 16-23), and additional elements required by the well-formedness rules of witnesses, e.g., the possible intermediate components from the C&C model between two components shown in the view. The component *lcpCmp* (l. 5) is the least common parent component in the C&C model that contains all components shown in the view. All subcomponents added to the witness are directly contained subcomponents in the C&C model. This also requires adding intermediate components from the C&C model to the witness (l. 8). All components added to the witness in the first step (ll. 5-9) have empty interfaces, i.e., have no ports.

Procedure 6 Witness generation for $m \vDash view$

```
 1: define witness as View
 2: define lcpCmp as Component
 3: define p′ as Port
 4: define chain as chain of Connector
 5: lcpCmp ← least common parent of view.Cmps in m
 6: add lcpCmp to witness
 7: for all cmp ∈ view.Cmps do
 8:    add cmp to witness with intermediate components
 9: end for
10: for all ac ∈ view.AbsCons do
11:    if ∄chain in witness with chain corresponds to ac then
12:       find chain in m with chain corresponds to ac
13:       add chain to witness with intermediate ports and components
14:    end if
15: end for
16: for all cmp ∈ view.Cmps do
17:    for all p ∈ cmp.Ports do
18:       if ∄p′ in witness with p′ corresponds to p then
19:          find p′ in m with p′ corresponds to p
20:          add p′ to cmp in witness
21:       end if
22:    end for
23: end for
24: return witness
```

To generate a small witness for $m \vDash view$, before adding a chain of connectors for an abstract connector or adding a port shown in the view to the witness, the algorithm checks whether the elements in the witness already provide a matching (ll. 11, 18). If no match of the view's element exists in the witness, a match in the C&C model is computed (ll. 12, 19) and added to the witness (ll. 13, 20). Please note that this check is a heuristics to create small witnesses, in terms of the number of concrete connectors and ports in the witnesses, but it does not guarantee that the generated witness is minimal to this measure. We discuss minimality of witnesses for satisfaction in Section 4.5.2.

Algorithm for generating witnesses for non-satisfaction

For every non-satisfaction reason found when checking the C&C model and view, the algorithm generates one witness that justifies non-satisfaction. The witnesses are generated according to the rules described in Section 4.3.2. All witnesses for non-satisfaction are unique. Thus, our algorithm always generates minimal witnesses with respect to the rules described in Section 4.3.2. We show the pseudo code for the algorithm in Procedure 7.

Missing Component The witness generated for a missing component is the empty witness together with the natural language description of the non-satisfaction reason

Procedure 7 Witness generation for $m \not\models view$

Missing Components $genWitnessMissingCmp(m, view, missingCmp)$

1: **define** *witness* **as** $View$
2: **add comment** $descrMissingCmp(m, view, missingCmp)$ **to** *witness*
3: **return** *witness*

Hierarchy Mismatch $genWitnessHierarchyMismatch(m, view, kind, cmp, subCmp)$

1: **define** *witness* **as** $View$
2: **add comment** $descrHierarchyMismatch(m, view, kind, cmp, subCmp)$ **to** *witness*
3: $lcpCmp \leftarrow$ **least common parent of** $\{cmp, subCmp\}$ **in** m
4: **add** *lcpCmp* **to** *witness*
5: **if** $lcpCmp \neq cmp$ **then**
6: **add** *cmp* **to** *witness* **with intermediate components**
7: **end if**
8: **add** *subCmp* **to** *witness* **with intermediate components**
9: **return** *witness*

Interface Mismatch $genWitnessInterfaceMismatch(m, view, kind, cmp, port)$

1: **define** *witness* **as** $View$
2: **add comment** $descrInterfaceMismatch(m, view, kind, cmp, port)$ **to** *witness*
3: **add** *cmp* **to** *witness*
4: **if** *kind* = PORT_NO_MATCH **then**
5: **for all** $p \in cmp.Ports$ **in** m **do**
6: **add** p **to** *cmp* **in** *witness*
7: **end for**
8: **else**
9: **add** *port* **from** m **to** *cmp* **in** *witness*
10: **end if**
11: **return** *witness*

Missing Connection $genWitnessMissingConnection(m, view, absConn)$

1: **define** *witness* **as** $View$
2: **define** *chain* **as chain of** $Connector$
3: **add comment** $descrMissingConnection(m, view, absConn)$ **to** *witness*
4: $lcpCmp \leftarrow$ **least common parent of** $\{absConn.srcCmp, absConn.tgtCmp\}$ **in** m
5: **add** *lcpCmp* **to** *witness*
6: **add** *absConn.srcCmp* **to** *witness*
7: **add** *absConn.tgtCmp* **to** *witness*
8: **if** *ac* **is port-to-port** $\vee$ **port-to-cmp then**
9: **for all** *chain* **in** m **with** *chain* **starts from** *ac.srcPort* **do**
10: **add** *chain* **to** *witness* **with intermediate ports and components**
11: **end for**
12: **else**
13: **for all** $p \in cmp.Ports$ **in** m **do**
14: **if** *p.direction* = IN **then**
15: **for all** *chain* **in** m **with** *chain* **starts from** p **do**
16: **add** *chain* **to** *witness* **with intermediate ports and components**
17: **end for**
18: **end if**
19: **end for**
20: **end if**
21: **return** *witness*

(see the first algorithm of Procedure 7).

Hierarchy Mismatch For all three kinds of hierarchy mismatches HIER_REV_CONT, HIER_IND_IN_CNCM, and HIER_IND_IN_VIEW the second algorithm in Procedure 7 computes the least common parent component of the conflicting components from the view and adds it to the witness (ll. 3-4). Only for the mismatch kind HIER_IND_IN_CNCM, the component *lcpCmp* is different from the component *cmp*. In this case, the algorithm adds the components *cmp* and its intermediate components from the C&C model to the witness (ll. 5-7). The algorithm then adds the component *subCmp* to the witness that demonstrates the contradicting containment. The algorithm also adds all intermediate components that exist in the C&C model between the components *lcpCmp* and *subCmp*.

Interface Mismatch For all three kinds of interface mismatches PORT_NO_MATCH, PORT_WRONG_DIR, and PORT_WRONG_TYPE the third algorithm in Procedure 7 adds the component with the interface mismatch to the witness (l. 3). In case there was no match (kind PORT_NO_MATCH) the algorithm adds all ports from the C&C model to the component in the view (ll. 4-7). In all other cases the single port from the C&C model that conflicts the port from the view is added to the witness.

Missing Connection The last algorithm in Procedure 7 creates a witness for a missing connection. The algorithm adds the source and target components together with their least common parent in the C&C model to the witness. In case the source port is known the algorithm adds all chains of connectors leaving the source port to the witness. In case the source port is unknown all chains of connectors leaving the source component are added. This demonstrates that there is no chain of connectors ending at the specified port of the target component (or any port of the target component if the target port is not given by the abstract connector).

4.3.4. Generating Natural Language Descriptions for Non-Satisfaction

In case C&C model does not satisfy a view we generate natural language descriptions that explain the non-satisfaction reasons. The texts are intended to help the engineer identifying the reasons for non-satisfaction.

To generate natural language descriptions for the reasons for non-satisfaction we use plain-text templates. The input of the templates is the output of the algorithm from Procedure 1 added to the set $NonSatReasons$. Each of the templates in Table 4.9, Table 4.10, Table 4.11, and Table 4.12 has a template name, a list of the inputs for the template, an optional block for assignments, and a block of the template text.

The template text is given in the translation notation defined in Appendix B. The notations consists of basic control structures that can be mixed with the resulting natural language text. Variables are printed in *italics*. Using a variable in the template text prints its value. Fixed parts of the resulting text are set in `underlined type writer font`.

Missing Component The natural language text for missing components is generated

from the template in Table 4.9. The inputs for this template are the C&C model m, the C&C view *view* and the component missing in the C&C model *missingCmp*.

Hierarchy Mismatch The natural language text template for a hierarchy mismatch is shown in Table 4.10. This template distinguishes the three different kinds of hierarchy mismatches and reports on the two components participating in the mismatch.

Interface Mismatch The natural language text template for missing connections is shown in Table 4.11. In this template we also report on the source and target port of the missing connection. Since port names are optional in C&C views we compute descriptions of the ports (*srcPort* and *tgtPort*) based on their availability in the `Assignments` section of the template.

Missing Connection The natural language text template for an interface mismatch is shown in Table 4.12. We distinguish three cases for port mismatches and generate natural language text containing the optional name and direction of the port from the C&C view that has no match in the C&C model.

Table 4.9.: Template for generating the natural language description for a missing component.

Template name	descrMissingCmp
Inputs	*m*, *view*, the missing component *missingCmp*
Text	`The component` *missingCmp.name* `of the view` *view.name* `is missing in the C&C model` *m.name* `.`

Table 4.10.: Template for generating the natural language description for a hierarchy mismatch.

Template name	descrHierarchyMismatch
Inputs	*m*, *view*, the mismatch kind HIER_REV_CONT, HIER_IND_IN_CNCM, or HIER_IND_IN_VIEW, the parent component *cmp*, and the child component *subCmp*

Text	**case** HIER_REV_CONT:
	`The component` *cmp.name* `contains the component` *subCmp.name* `in the view` *view.name* `but the latter contains the former in the C&C model` *m.name* `.`
	
	case HIER_IND_IN_CNCM:
	`The component` *cmp.name* `and the component` *subCmp.name* `are independent in the view` *view.name* `but not in the C&C model` *m.name* `.`
	
	case HIER_IND_IN_VIEW:
	`The component` *cmp.name* `and the component` *subCmp.name* `are independent in the C&C model` *m.name* `but not in the view` *view.name* `.`

Table 4.11.: Template for generating the natural language description for a missing connection.

Template name	descrMissingConnection
Inputs	*m*, *view*, and the abstract connector *ac*
Assignments	*srcPort* ← **if** *ac.srcPort* = ⊥ **then** `"unnamed port"` **else** `"port"` +*ac.srcPort.name* *tgtPort* ← **if** *ac.tgtPort* = ⊥ **then** `"unnamed port"` **else** `"port"` +*ac.tgtPort.name*
Text	`The C&C model` *m.name* `is missing a connection from the component` *ac.srcCmp* `to the component` *ac.tgtCmp* `(from` *srcPort* `to` *tgtPort*`)` `.`

Table 4.12.: Template for generating the natural language description for an interface mismatch.

Template name	descrInterfaceMismatch
Inputs	m, *view*, the mismatch kind PORT_NO_MATCH, PORT_WRONG_DIR, or PORT_WRONG_TYPE, the component *cmp*, and the port p
Assignments	$pName \leftarrow$ **if** $p.name = \bot$ **then** `"unnamed port"` **else** $p.name$ $direction \leftarrow$ **if** $p.direction =$ `IN` **then** `"incoming"` **else** `"outgoing"`

Text	**case** PORT_NO_MATCH:
	`No match for port` $pName$ `of the component` $cmp.name$ `.`
	
	case PORT_WRONG_DIR:
	`Wrong direction for port` $pName$ `of the component` $cmp.name$ `(` $direction$ `).`
	
	case PORT_WRONG_TYPE:
	`Wrong type for port` $pName$ `of the component` $cmp.name$ `(` $p.type$ `).`

4.3.5. Correctness and Completeness

We show the correctness and completeness of the algorithm in Procedures 1-5 for determining whether an C&C model m satisfies a view *view* and for computing the reasons for non-satisfaction. Correctness means that whenever the algorithm computes at least one non-satisfaction reason, indeed $m \not\models view$. Completeness means that whenever $m \not\models view$, the algorithm computes at least one reason for non-satisfaction.

Lemma 4.13. The verification algorithm in Procedures 1-5 is correct, i.e.,

$$computeNonSatReasons(m, view) \neq \varnothing \Rightarrow m \not\models view.$$

We show the correctness of Procedures 1-5 according to Lemma 4.13 with respect to Definition 3.8 of the satisfaction $m \models view$. We show that every element added to the set $NonSatReasons$ in Procedures 2-5 executed by the algorithm is a reason for non-satisfaction of the C&C model and view. The proof — as the algorithm itself — is structured by the four classes of reasons for non-satisfaction and the lines from Procedures 2-5 that add elements to the set $NonSatReasons$. Most proof steps work by assuming satisfaction and then deriving contradictions with well-formedness rules or properties of the elements identified by the algorithm.

Proof. We show that whenever the algorithm adds an element to the set $NonSatReasons$, the C&C model m does indeed not satisfy the view *view*.

Missing Components (Procedure 2, l. 3) From the existence of component $missingCmp \in (view.Cmps \setminus m.Cmps)$ it follows that $view.Cmps \not\subseteq m.Cmps$. This contradicts Definition 3.8, Item 3, thus $m \not\models view$.

Hierarchy Mismatch (Procedure 3, ll. 6, 8) According to line 3 component $subCmp$ is a subcomponent of component cmp in the view: $subCmp \in view.subs(cmp)$. Assuming $m \models view$ it follows from Definition 3.8, Item 3 that $subCmp \in m.subs^+(cmp)$. This contradicts $subCmp \notin m.subs^+(cmp)$ from line 4 and the well-formedness rules of the C&C model from Definition. 2.2, thus $m \not\models view$.

Hierarchy Mismatch (Procedure 3, ll. 15, 17) Component cmp' is a subcomponent of component cmp (l. 12): $cmp' \in view.subs(cmp)$. Assuming $m \models view$ it follows from Definition 3.8, Item 3 that $cmp' \in m.subs^+(cmp)$. This contradicts $subCmp \notin m.subs^+(cmp)$ from line 13 and the well-formedness rules of the C&C model from Definition. 2.2, thus $m \not\models view$.

Interface Mismatch (Procedure 4, l. 6) Assuming $m \models view$ it follows from the existence of a port $p \in view.ports(cmp)$ (l.3) by Definition 3.8, Item 4 (a) that $\exists p' \in m.ports(cmp) : p \cong p'$ which contradicts $\nexists p' \in m.ports(cmp) : view.dir(p) =$

$m.dir(p') \wedge view.type(p) \in \{m.type(p'), \bot\}$ (l. 5). Thus, if the algorithm adds an element to $NonSatReasons$ in l. 6 the C&C model indeed does not satisfy the view.

Interface Mismatch (Procedure 4, l. 9) Assuming $m \vDash view$ it follows from the existence of a port $p \in view.ports(cmp)$ (l.3) by Definition 3.8, Item 4 (a) that $\exists p' \in m.ports(cmd) : p \cong p'$ and from $view.name(p) \neq \bot$ (**`else if`** of condition in line 4) by Definition 3.8, Item 4 (b3) that $view.name(p) = m.name(p')$ which contradicts $view.name(p) \neq m.name(p')$ (l. 8). Thus, if the algorithm adds an element to $NonSatReasons$ in line 9 the C&C model indeed does not satisfy the view.

Interface Mismatch (Procedure 4, l. 13) Assuming $m \vDash view$ it follows from the existence of a port $p \in view.ports(cmp)$ (l.3) by Definition 3.8, Item 4 (a) that $\exists p' \in m.ports(cmd) : p \cong p'$ and from line 11 by Definition 3.8, Item 4 (b1) that $view.dir(p) = m.dir(p')$ which contradicts $view.dir(p) \neq m.dir(p')$ (l. 12). Thus, if the algorithm adds an element to $NonSatReasons$ in line 13 the C&C model indeed does not satisfy the view.

Interface Mismatch (Procedure 4, l. 16) Assuming $m \vDash view$ it follows from the existence of a port $p \in view.ports(cmp)$ (l.3) by Definition 3.8, Item 4 (a) that $\exists p' \in m.ports(cmd) : p \cong p'$ and from line 11 by Definition 3.8, Item 4 (b2) that $view.type(p) \in \{\bot, m.type(p')\}$ which contradicts $view.type(p) \neq \bot \wedge view.type(p) \neq m.type(p')$ (l.15). Thus, if the algorithm adds an element to $NonSatReasons$ in l.16 the C&C model indeed does not satisfy the view.

Missing Connection (Procedure 5, ll. 4, 6, 8, 10) Assuming $m \vDash view$, the existence of a chain of connectors $c_1, ..., c_n \in m.Cons$ follows from Definition 3.8, Item 5.

- The connector c_1 starts from the component $ac.srcCmp$ (Definition 3.8, Item 5 (a)) at the port $c_1.srcPort$ that by definition of the C&C model belongs to the component $c_1.srcCmp = ac.srcCmp$ (Definition 2.2, Item 3).
- The connector c_n connects to component $ac.tgtCmp$ (Definition 3.8, Item 5 (b)) via port $c_n.tgtPort$ that by definition of the C&C model belongs to component $c_n.tgtCmp = ac.tgtCmp$ (Definition 2.2, Item 3).

The satisfaction of the condition in line 3 with $ac.srcPort = \bot = ac.tgtPort$ and $p := c_1.srcPort$ and $p' := c_n.tgtPort$ leads to a contradiction of the assumption, since p and p' exist in the C&C model. Thus, if the algorithm adds an element to $NonSatReasons$ in line 4 the C&C model indeed does not satisfy the view.

The satisfaction of the condition in line 5 with $ac.srcPort = \bot \neq ac.tgtPort$ and $p := c_1.srcPort$ and $p' := c_n.tgtPort$ leads to a contradiction of the assumption, since p and p' exist in the C&C model and $ac.tgtPort \cong p'$ (Definition 3.8, Items 5 (b)).

Thus, if the algorithm adds an element to $NonSatReasons$ in line 6 the C&C model indeed does not satisfy the view.

The satisfaction of the condition in line 7 with $ac.srcPort \neq \bot = ac.tgtPort$ and $p := c_1.srcPort$ and $p' := c_n.tgtPort$ leads to a contradiction of the assumption, since p and p' exist in the C&C model and $ac.srcPort \cong p$ (Definition 3.8, Items 5 (a)). Thus, if the algorithm adds an element to $NonSatReasons$ in line 8 the C&C model indeed does not satisfy the view.

The satisfaction of the condition in line 9 with $ac.srcPort \neq \bot \neq ac.tgtPort$ and $p := c_1.srcPort$ and $p' := c_n.tgtPort$ leads to a contradiction of the assumption, since p and p' exist in the C&C model, $ac.srcPort \cong p$ (Definition 3.8, Items 5 (a)), and $ac.tgtPort \cong p'$ (Definition 3.8, Items 5 (b)). Thus, if the algorithm adds an element to $NonSatReasons$ in line 10 the C&C model indeed does not satisfy the view.

Every addition of elements to the initially empty set $NonSatReasons$ means that the C&C model m does not satisfy the view $view$ and thus:

$$computeNonSatReasons(m, view) \neq \varnothing \Rightarrow m \not\models view.$$

□

We now state and prove completeness of the algorithm from Procedures 1-5: if a C&C model does not satisfy a view, the algorithm will find at least one reason for non-satisfaction.

Lemma 4.14. The verification algorithm in Procedures 1-5 is complete, i.e.,

$$m \not\models view \Rightarrow computeNonSatReasons(m, view) \neq \varnothing.$$

Proof. According to Definition 3.8 $m \models view$ if and only if all properties of m and $view$, as stated in items 1-5, hold. Thus, for $m \not\models view$ it is enough that one of the properties described in items 1-5 does not hold. We show for every case of non-satisfaction that the algorithm detects this case and adds at least one element to the set $NonSatReasons$. The proof assumes for every required property the negation of the property and shows that a non-satisfaction reason is generated by the algorithm.

Detecting violation of Definition 3.8, Item 1 Assume $view.Types \not\subseteq m.Types$:

Since every type $missingType \in (view.Types \setminus m.Types)$ appears on a port in the view (see Definition 3.6, Item 4) this implies $\exists p \in view.Ports$ such that $view.type(p) = missingType \neq \bot$. In addition, $missingType \notin m.Types$ together with Definition 2.2, Item 4 implies $\forall p' \in m.Ports : m.type(p') \neq missingType$.

This case is thus analogous to the item [Detecting violation of Definition 3.8, Item 4] of this proof since for the port p with $view.type(p) = missingType$ none of the ports $p' \in m.Ports$ satisfy $p \cong p'$. The algorithm adds an element to $NonSatReasons$ in Procedure 4.

Detecting violation of Definition 3.8, Item 2 Assume $view.Cmps \not\subseteq m.Cmps$:
This implies the existence of at least one component $missingCmp \in (view.Cmps \setminus m.Cmps)$. The condition in Procedure 2, line 2 is satisfied. The algorithm will add an element to $NonSatReasons$ in line 3.

Detecting violation of Definition 3.8, Item 3 Assume $\exists cmp_1, cmp_2 \in view.Cmps$ with $cmp_1 \in view.subs(cmp_2)$ and $cmp_1 \notin m.subs^+(cmp_2)$:

- In case cmp_1 or $cmp_2 \notin m.Cmps$ the algorithm will add an element to $NonSatReasons$ for the missing component as in the item [Detecting violation of Definition 3.8, Item 2].
- For $cmp_1, cmp_2 \in m.Cmps$ the nested loops in Procedure 3 in line 2 and line 3 will be entered with $cmp := cmp_2$ satisfying $cmp \in view.Cmps \cap m.Cmps$ (l. 2) and $subCmp := cmp_1$ satisfying $subCmp \in view.subs(cmp)$ (l. 3).

  The condition in line 4 is exactly the assumption $cmp_1 \notin m.subs^+(cmp_2)$ and thus the algorithm adds an element to $NonSatReasons$ in either line 6 or line 8.

Detecting violation of Definition 3.8, Item 4 Assume $\exists cmp' \in view.Cmps$ $\exists p_1 \in view.ports(cmp') \forall p_2 \in m.ports(cmp') : p_1 \not\cong p_2$:

- For $cmp' \notin m.Cmps$ see item [Detecting violation of Definition 3.8, Item 2].
- Otherwise the algorithm will enter the nested loops in Procedure 4 in line 2 and line 3 with $cmp := cmp'$ and $p := p_1$.

  In case $p_1.name = \bot$ the assumption implies that either $m.ports(cmp') = \varnothing$ or that with Definition 3.8, Item 4 (b) of $\cong$ unfolded $\forall p_2 \in m.ports(cmp') : view.dir(p) \neq m.dir(p') \vee view.type(p) \notin \{m.type(p'), \bot\}$. In both cases the condition in line 5 is satisfied and the algorithm adds a reason for non-satisfaction to the set $NonSatReasons$ in line 6.

  In case $p_1.name \neq \bot$ the assumption implies that either $m.ports(cmp') = \varnothing$ or that for all ports $p_2 \in m.ports(cmp)$ at least one of the conditions from Definition 3.8, Item 4 (b1)-(b3) of $p_1 \cong p_2$ is violated.

  - In case of $m.ports(cmp') = \varnothing$ the condition in line 8 is satisfied and the algorithm adds a reason for non-satisfaction to the set $NonSatReasons$ in line 9.
  - Otherwise we have left two cases of possible violations for $p_1 \cong p_2$ for all ports $p_2 \in m.ports(cmp)$. In case $view.name(p_1) \in \{\bot, m.name(p_2)\}$ (Definition 3.8, Item 4 (b3)) and $p_1.name \neq \bot$ (see above) we obtain $\forall p_2 \in m.ports(cmp') : m.name(p_2) \neq view.name(p_1)$ which satisfies the condition in line 8 with $cmp = cmp'$ and $p = p_1$. The algorithm adds a reason for non-satisfaction to the set $NonSatReasons$ in line 9.

In all other cases there is a unique port $p_2 \in cmp.ports(cmp')$ with $m.name(p_2) = view.name(p_1)$. The algorithm assigns this port to the variable p' in line 11.

In case Definition 3.8, Item 4 (b1) is not satisfied, the condition in line 12 is met and the algorithm adds a reason for non-satisfaction to the set $NonSatReasons$ in line 13.

In case Definition 3.8, Item 4 (b2) is not satisfied, the condition in line 15 is met and the algorithm adds a reason for non-satisfaction to the set $NonSatReasons$ in line 16.

Detecting violation of Definition 3.8, Item 5 (a) Assume $\exists ac \in view.AbsCons$ $\forall c_1, ..., c_n \in m.Cons : \neg(ac.srcCmp = c_1.srcCmp) \vee \neg(ac.srcPort \cong c_1.srcPort \vee ac.srcPort = \perp)$:

- The case $\neg(ac.srcCmp = c_1.srcCmp)$ means that there is no connector in the C&C model starting from component $ac.srcCmp$. Thus no chain of connectors can exist and any condition in Procedure 5, lines 3-9 evaluates to true. The algorithm will add an element to the set $NonSatReasons$.
- The case $\neg(ac.srcPort \cong c_1.srcPort \vee ac.srcPort = \perp)$ implies that port $ac.srcPort \neq \perp$ is known but there exists no connector in the C&C model starting from a port that is equivalent to $ac.srcPort$ and that is a port of the component $ac.srcCmp$. Thus no chain of connectors can exist and the two conditions in Procedure 5 in line 7 and in line 9 evaluate to true. The algorithm will add an element to the set $NonSatReasons$.

Detecting violation of Definition 3.8, Item 5 (b) Assume $\exists ac \in view.AbsCons$ $\forall c_1, ..., c_n \in m.Cons : \neg(ac.tgtCmp = c_n.tgtCmp) \vee \neg(ac.tgtPort \cong c_1.tgtPort \vee ac.tgtPort = \perp)$:

- The case $\neg(ac.tgtCmp = c_n.tgtCmp)$ means that there is no connector connecting to component $ac.tgtCmp$ thus no chain of connectors can exist and any condition in Procedure 5 in lines 3-9 evaluates to true. The algorithm will add an element to the set $NonSatReasons$.
- The case $\neg(ac.tgtPort \cong c_1.tgtPort \vee ac.tgtPort = \perp)$ implies that the port $ac.tgtPort \neq \perp$ is known in the C&C view but there exists no connector in the C&C model connecting to a port equivalent to port $ac.tgtPort$ that is a port of the component $ac.tgtCmp$. Thus no chain of connectors can exist and any conditions in Procedure 5 in lines 5 and 9 evaluate to true. The algorithm will add an element to the set $NonSatReasons$.

Detecting violation of Definition 3.8, Item 5 (c) Assume $\exists ac \in view.AbsCons$ $\forall c_1, ..., c_n \in m.Cons : \neg(\forall 1 \leq i < n : c_i.tgtPort = c_{i+1}.srcPort)$:

- The assumption $\neg(\forall 1 \leq i < n : c_i.tgtPort = c_{i+1}.srcPort)$ for all possible chains $c_1, ..., c_n$ is equivalent to $\exists 1 \leq i < n : c_i.tgtPort \neq c_{i+1}.srcPort$. Thus, no connected chain of connectors exists in the C&C model and any condition from Procedure 5 in lines 3-9 evaluates to true. The algorithm will add an element to the set $NonSatReasons$.

Every violation of Definition 3.8 for $m \vDash view$ will be detected by the algorithm and results in adding at least one element to the set $NonSatReasons$ of reasons for non-satisfaction and thus:

$$m \nvDash view \Rightarrow computeNonSatReasons(m, view) \neq \varnothing.$$

□

4.3.6. Complexity

We now examine the algorithm in Procedures 1-5, the time complexity and the maximal number of reasons for non-satisfaction it computes. The time complexities of the algorithms for each check for non-satisfaction reasons are summarized in Table 4.15.

Non-Satisfaction Check	Time Complexity
Missing Component	$O(\|view.Cmps\|)$
Hierarchy Mismatch	$O(\|view.Cmps\|^2 * \|m.Cmps\|)$
Interface Mismatch	$O(\|view.Ports\| * \|m.Ports\|)$
Missing Connection	$O(\|view.AbsCons\| * \|m.Ports\| * \|m.Cons\|)$

Table 4.15.: Time complexities for the checks for non-satisfaction reasons of the algorithm from Procedures 1-5. The inputs are a C&C model m and a C&C view $view$. These are the time complexities of our implementation and thus only upper bounds for the complexity of the satisfaction checking problem.

Time complexity

The time complexity of the algorithm in Procedures 1-5 depends on the size of the two input structures m and $view$. It depends on the number of components in the C&C model $|m.Cmps|$, its number of ports $|m.Ports|$ and its number of connectors $|m.Cons|$. It also depends on the number of components in the view $|view.Cmps|$, its number of ports $|view.Ports|$ and its number of abstract connectors $|m.AbsCons|$.

We examine the time complexity for each procedure called from Procedure 1 separately. These results describe the complexity of our algorithm and thus only give an upper bound to the complexity of the satisfaction problem for C&C views.

Missing Component (Procedure 2) We consider the look up $c \in m.Cmps$ to take constant time. The computation time for non-satisfaction reasons is thus linear in $|view.Cmps|$.

Hierarchy Mismatch (Procedure 3) The check of c_1 **is parent of** c_2 requires at most enumerating all the (indirectly contained) subcomponents of c_1. The time complexity of our current naive implementation of checking c_1 **is parent of** c_2 **in** m is in $O(|m.Cmps|)$, thus the time complexity of the Hierarchy Mismatch part of the algorithm is in $O(|view.Cmps|^2 * |m.Cmps|)$. It could be reduced to $O(|m.Cmps|^2 + |view.Cmps|^2)$ by employing a pre-processing step of the **subs** relation that will allow the algorithm to determine c_1 **is parent of** c_2 **in** m in constant time.

Interface Mismatch (Procedure 4) Every port in the view is checked for a corresponding port in the C&C model. For each unnamed port the algorithm checks all ports of the corresponding component in the C&C model for a matching type and direction. The computation time for the non-satisfaction reasons is thus in $O(|view.Ports| * |m.Ports|)$.

Missing Connection (Procedure 5) The computation of reasons for non-satisfaction requires, in the worst case, for all abstract connectors to look at all possible source ports in the C&C model and all outgoing chains of connectors. The time complexity is thus in $O(|view.AbsCons| * |m.Ports| * |m.Cons|)$.

Maximal number of reasons for non-satisfaction

For every class of reasons for non-satisfaction the algorithm in Procedures 1-5 adds a bounded number of elements to $NonSatReasons$. For Missing Component, the algorithm adds at most $|view.Cmps|$ elements to $NonSatReasons$. For Hierarchy Mismatch, every pair of components from the view can only be in one of the four containment relations from Procedure 3 lines 5, 7, 14, 16. Thus no more than $|view.Cmps|^2$ elements are added to the set $NonSatReasons$. For Interface Mismatch, the algorithm adds at most $2 * |view.Ports|$ elements to $NonSatReaons$ in case all ports in the view have wrong type and direction. For Missing Connection, the algorithm adds at most $|view.AbsCons|$ elements to $NonSatReasons$ because the four cases it considers are disjoint.

Please note that these upper bounds are not independent. For example, if the algorithm computes $|view.Cmps|$ reasons of non-satisfaction because of missing components no further reasons for non-satisfaction will be found.

4.4. Implementation and Evaluation

We have implemented support for C&C views within a prototype Eclipse plug-in, on top of MontiCore [wwwv, KRV10]. MontiCore provides parsers, Eclipse editors etc. for MontiArc [wwwq, HRR12].

Our implementation is completely written in Java. It consists of a library providing utility operations on C&C views and C&C models including a bridge from and to MontiCore ASTs, a verification engine implementing the verification algorithms, and an Eclipse plug-in. We report the size of the implementation in effective lines of code

(ELOC). Lines counted as ELOC contain characters other than white spaces or comments and are contained in classes of the implementation. The numbers of ELOC do neither include unit tests nor code for validation. The C&C views and C&C model utility library consists of 9 classes with a total of 911 ELOC. The C&C views verification engine consists of 19 classes with a total of 1,129 ELOC. The C&C views verification plug-in consists of 11 classes with a total of 565 ELOC.

Using the C&C views verification plug-in, the engineer can select two files, an C&C model and a view, and check whether the C&C model satisfies the view. In case of a positive answer, a witness for satisfaction is generated and presented as an annotated view in the main editing pane. In case of a negative answer, all witnesses for non-satisfaction are generated and listed in a hierarchical problems view titled *Witnesses for Non-Satisfaction*. The hierarchy in the problems view reflects the classification of non-satisfaction reasons described above, i.e., of missing component, hierarchy mismatch, interface mismatch, and missing connection. Clicking a specific entry in the problems view opens up the corresponding witness as an annotated view in the main editing pane. Figure 4.16 shows a screen capture from the prototype plug-in.

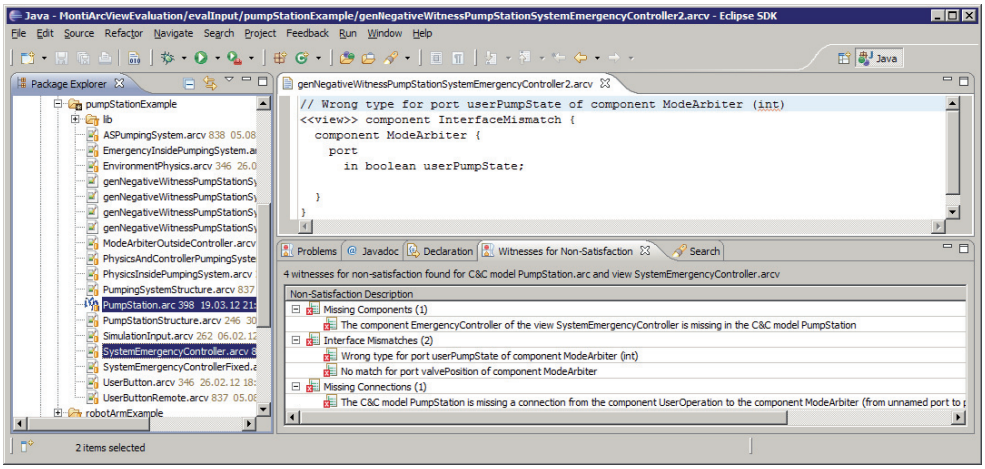


Figure 4.16.: A screen capture from the prototype plug-in, after checking the C&C model `PumpStation` shown in Figure 4.1 against the view `SystemEmergencyController` shown in Figure 4.3 (b). The lower pane shows the Eclipse problems view titled *Witnesses for Non-Satisfaction*, which provides a hierarchical list of the generated witnesses for non-satisfaction together with their generated natural language descriptions. Four witnesses were generated, one for a missing component, two for interface mismatches, and one for a missing connection. The main editing pane on the top right shows one of the generated witnesses for interface mismatch.

We present a tutorial on how to install the plug-in, import the example systems, and execute C&C views verification in Appendix C.

4.4.1. Example Systems

We evaluated our work on C&C models and views taken from four example systems, from different sources and of different domains. The evaluation on example systems is of qualitative nature. We wanted to gain experience with creating C&C views for documenting and specifying C&C models and their use for C&C views verification. We describe the C&C models and the C&C views that we have created below and report on the lessons learned in Section 4.4.4.

All the C&C models, views, and specifications used in our evaluation, as described below, are available with the prototype tool implementation as supporting materials from [wwwu]. We encourage the interested reader to inspect them.

Avionics system

We evaluated our work on an AADL architecture of an avionics system. Specifically, the model `Avionics_System.aadl` of the OSATE AADL Project, available from [wwwa]. The avionics system architecture is a high-level model of several avionics system subsystems. The purpose of this AADL architecture is to determine the minimum end-to-end flow latency for providing a new page on a multi-function display. Indirect AADL flows between the `Pilot_Display` component and the `Flight_Director` component are specified with minimal latencies. Since in this work we are only interested in the structure of the C&C models, in our translation of this AADL architecture into a MontiArc C&C model we have ignored the flows definitions but preserved the hierarchical structure and all ports and connectors.

The avionics system C&C model has 6 components, 16 ports, and 8 connectors. The depth of the component hierarchy of the system architecture is 2.

Based on various use cases, related to the interaction between the avionics system's components, we created 9 C&C views, with 1-6 components each. The views are inspired by the original analysis of AADL's flows. For example, one view gives an overview of the complete data flow in the system, declared using abstract connectors. This view does not provide additional information such as port names or types. Another view provides more details about the communication between the `Pilot_Display` and it's `Page_Content_Manager`, showing incoming and outgoing ports with their names and connectors. An overview of the C&C views and their sizes is given in Table 4.17. The table lists all views, their number of components and abstract connectors, satisfaction by the C&C model, the number of generated witnesses and satisfaction checking and witness generation time in milliseconds.

We defined 2 views specifications, one with 5 views and one with 9 views. The first deals only with the views that contain components that are connected to the `Pilot_Display`. The second contains all the views that we have created.

Checking satisfaction produced a positive witness in 6 of 9 cases. The algorithm produced up to 4 witnesses for non-satisfaction for each C&C view not satisfied by the C&C model (see Table 4.17).

Bumper bot

We evaluated our work on the software architecture of a Lego Mindstorms NXT [wwwl] bumper bot similar to the bumper car model from [wwwx]. The bumper bot can power its left and right motors and detect obstacles in front of it. In addition, it is equipped with an emergency stop button. The bumper bot's objective is to go around obstacles and keep driving forward. As part of this case study system, we have designed a set of views and a complete C&C model for the bumper bot software in MontiArc. We have used automata to define the behavior of the components mentioned in the bumper bot C&C model, used the code generator described in Chapter 8, and successfully deployed the generated code to the controller of the bumper bot.

The bumper bot architecture consists of 12 components, 28 ports, and 20 connectors. It is a layered architecture with three layers, the sensors layer, the control components layer, and the actuators layer. The depth of the component hierarchy of the bumper bot C&C model is 3.

We developed 8 C&C views in total, each with 2-8 components. For example, one view shows the basic components of the bumper bot, showing neither the layers of the architecture nor the emergency stop related components. Another view contains only the structure and division into layers, without any ports or connectors. Of the 8 C&C views, 5 are independent of the emergency stop feature and describe only the components and abstract connectors for fulfilling the main purpose of the bumper bot. The remaining 3 views exhibit components required by the emergency stop feature. For example, one of these adds a mode arbiter to the components and abstract connectors participating in the regular robot control. Another one shows the components and paths of signals used in case of an emergency stop, including the mode arbiter. An overview of the C&C views and their sizes is given in Table 4.17. The table lists all views, their number of components and abstract connectors, satisfaction by the C&C model, the number of generated witnesses and satisfaction checking and witness generation time in milliseconds.

Checking satisfaction produced a positive witness in 6 of 8 cases. The algorithm produced up to 2 witnesses for non-satisfaction for each C&C view not satisfied by the C&C model (see Table 4.17).

Pump station

We further evaluated our work on a pump station C&C model taken from an example system provided with the AutoFOCUS tool [BHS99, HF07, wwwe] (the C&C model we use as a running example introduced in Section 3.1). The physical pump station system consists of two water tanks connected by a pipeline system with a valve and a pump. The water level in the first water tank can rise (this is controlled by the environment). When

the water level of the first tank rises to a critical level, the water has to be pumped to the second water tank. The second water tank has a drain. The C&C model presented in Figure 4.1 also shows a model of the environment with a physics simulation, used to test the pumping system.

The pump station C&C model consists of 16 components, 67 ports, and 47 connectors. The depth of the component hierarchy of the pump station C&C model is 4.

Again, based on several design decisions and relations we wanted to highlight and document, we created 11 C&C views, each with 2-5 components. For example, one view gives an overview of the basic structure of the system and omits details about interfaces and connectors. Another view documents part of the connections between the actuators and their environment, hiding hierarchies and omitting elements not connected to the actuators. An additional C&C view shows an undesired design where the simulation component is placed inside the pumping system. We have already described some of the views of this case study in Section 4.1. An overview of the C&C views and their sizes is given in Table 4.17. The table lists all views, their number of components and abstract connectors, satisfaction by the C&C model, the number of generated witnesses and satisfaction checking and witness generation time in milliseconds.

To be able to document undesired designs and alternatives, we created 7 views specifications. Two specifications specify the optional existence of an emergency system and its implications (one of these is specification S_1 described in Section 3.5). Another specification prohibits an emergency system. Further specifications combine views of the function of the pump station with ones that specify the separation of the pumping system from the simulation part.

Checking satisfaction produced a positive witness in 6 of 11 cases. The algorithm produced up to 4 witnesses for non-satisfaction for each C&C view not satisfied by the C&C model (see Table 4.17).

Robotic arm

Finally, we evaluated C&C views verification on a robotic arm C&C model — specifically the rotational joint of a robotic arm, taken from an industrial system by VTT Tampere, Finland. The main components of the rotational joint's C&C model are a cylinder, a servo valve, a sensor, a joint limiter, and an actuator. The rotational joint is a subsystem of a robotic arm containing in total eight identical copies of rotational and translational joints.

The robotic arm rotational joint C&C model consists of 8 components, 18 ports, and 16 connectors. The depth of the component hierarchy of the robotic arm rotational joint C&C model is 3.

Again, based on several requirements and partial knowledge or particular features, we created 11 C&C views, each with 1-5 components. Some views highlight the components necessary for the function of the joint while others document design alternatives on the placement of sensor and actuator components. Some of the views give an overview of related components with only few details of their interfaces or connectedness. Other views document complete interfaces of relevant components and some of their connec-

tions. An overview of the C&C views and their sizes is given in Table 4.17. The table lists all views, their number of components and abstract connectors, satisfaction by the C&C model, the number of generated witnesses and satisfaction checking and witness generation time in milliseconds.

To experiment with non-satisfaction, we modified the C&C model to not satisfy any of the views. The modifications are renaming ports, removing connectors, and changing component hierarchies. The algorithm produced up to 4 witnesses for non-satisfaction for each C&C view not satisfied by the C&C model (see Table 4.17).

Running times on example systems

We have measured the running times of our algorithm for the C&C views and example systems presented above. We executed all the experiments on an ordinary laptop computer, Intel Dual Core CPU, 2.8 GHz, running 64-bit Windows 7 and Java 1.7.0_17. We repeated the experiments 12 times for each C&C model and view.

Table 4.17 reports on the numbers of components and connectors of each C&C model, the numbers of components and abstract connectors of each view, the satisfaction result, the number of generated witnesses, the time for checking satisfaction and computing non-satisfaction reasons (t_c) in milliseconds, and the time for generating the witnesses (t_g) in milliseconds. The times were measured by the system clock accessed through the Java API and are presented in the table as the average computed over 12 runs.

It is interesting to see that the average verification time stays below a millisecond. The four cases where the verification time is 0.5ms or 1ms are half cases of satisfaction and non-satisfaction. The maximum time of 8.5ms for witness generation is spent in case of non-satisfaction.

Threats to Validity

The choice of C&C models for our experiments is limited to four systems from different sources. To address the threat to generalizability of the results, we have selected example systems from different domains: avionics, robotics, automation, and control. Further studies with more and larger real-world systems could provide more insight and might allow capturing differences and similarities between the C&C models of diverse domains.

For all example systems only C&C models were available and we created the C&C views ourselves. An evaluation with independent subjects to assess the expressiveness and comprehensibility of C&C views could address this threat of a possible bias.

Please note that the running times for C&C views verification reported in Table 4.17 are very low. We do not use these to asses the performance and scalability of our algorithms. We instead set up different experiments to analyze performance and scalability.

4.4.2. Performance and Scalability

To evaluate the performance and scalability of C&C views verification in handling large C&C models and views, we have experimented with synthesized C&C models of differ-

Model	C	Con	View	C	ACon	SAT	Wit	t_c	t_g
AvionicsSystem	6	8	ConnectPilotDisplayAndPCM	2	1	✗	1	0	2
AvionicsSystem	6	8	ControlFlowInSystem	5	8	✓	1	0	0
AvionicsSystem	6	8	DisplayAndManager	2	2	✓	1	0	0
AvionicsSystem	6	8	FlightManagerAndDirector	2	2	✓	1	0	0
AvionicsSystem	6	8	FlightSystemStructure	6	0	✓	1	0	0
AvionicsSystem	6	8	PilotAndPageContentManager	3	2	✓	1	0	0
AvionicsSystem	6	8	PDMIndependentOfPilotDisplay	2	0	✓	1	0	0
AvionicsSystem	6	8	PDMInsidePilotDisplay	2	0	✗	1	0	2
AvionicsSystem	6	8	PDMPortsReversed	1	0	✗	4	0	5
BumperBot	12	18	BumpControlOverview	3	3	✓	1	0	0
BumperBot	12	18	BumperBotEmgSystem	3	1	✓	1	0	0
BumperBot	12	18	BumperBotSensors	2	0	✓	1	0	0
BumperBot	12	18	BumperBotStructure	8	5	✗	2	1	4.5
BumperBot	12	18	BumperBotStructureOnly	8	0	✓	1	0	0
BumperBot	12	18	BumperBotMotorWrongPlace	3	0	✗	2	0	3
BumperBot	12	18	MotorArbiterConnectionsEmergency	5	4	✓	1	0	0
BumperBot	12	18	MotorArbiterConnectionsOldBehavior	4	4	✓	1	0	0
PumpStation	16	47	ASPumpingSystem	5	2	✓	1	0	0
PumpStation	16	47	EnvironmentPhysics	4	4	✓	1	0.5	0
PumpStation	16	47	ModeArbiterOutsideController	2	0	✗	1	0	1.5
PumpStation	16	47	PhysicsAndControllerPumpingSystem	3	1	✗	2	0.5	6
PumpStation	16	47	PhysicsInsidePumpingSystem	2	0	✗	1	0	2
PumpStation	16	47	PumpStationStructure	3	0	✓	1	0	0
PumpStation	16	47	PumpingSystemStructure	5	5	✓	1	1	0
PumpStation	16	47	SimulationInput	3	3	✓	1	0	0
PumpStation	16	47	SystemEmergencyController	4	3	✗	4	0	5.5
PumpStation	16	47	SystemEmergencyControllerFixed	4	3	✗	1	0	1
PumpStation	16	47	UserButton	4	4	✓	1	0	0
RotationalJoint	8	16	ASDependence	3	0	✗	1	0	1.5
RotationalJoint	8	16	BodySensorIn	5	5	✗	4	0	8.5
RotationalJoint	8	16	BodySensorOut	5	5	✗	3	0	6
RotationalJoint	8	16	OldDesign	3	1	✗	3	0	5
RotationalJoint	8	16	OldDesignExternalCylinder	3	1	✗	2	0	4
RotationalJoint	8	16	RJFunction	4	1	✗	1	0	2
RotationalJoint	8	16	RJStructure	4	4	✗	2	0	4
RotationalJoint	8	16	SensorAmplifierView	1	0	✗	1	0	1
RotationalJoint	8	16	SensorConnections	3	2	✗	1	0	2
RotationalJoint	8	16	SensorConnectionsInterfaceComplete	3	2	✗	1	0	2
RotationalJoint	8	16	SensorHasAmplifier	2	0	✗	1	0	1

Table 4.17.: C&C views verification times for four example systems. The table reports on the numbers of components and connectors of the C&C model, the numbers of components and abstract connectors of the view, the satisfaction result, the number of generated witnesses, the time for checking satisfaction (t_c) in milliseconds, and the time for generating the witnesses (t_c) in milliseconds. Times are reported as averages computed over 12 runs.

ent sizes and with related synthesized views where we have randomly applied various mutations listed in Table 4.18. We describe the setup of our experiments below. The code to reproduce our experiments or further define and execute similar ones is available together with related evaluation materials for C&C views verification from [wwwu]. Appendix C explains how to use the provided code and execute the experiments.

We have implemented a generator for randomly constructed C&C models. The generator constructs C&C models based on the following parameters: number of components, maximal number of subcomponents per component, number of types of ports in the C&C model, maximal number of ports in the C&C model, and maximal number of connectors in the C&C model.

The algorithm first creates the root component of the C&C model. Then it creates new components and adds them as subcomponents to existing components in the C&C model until the total number of components equals the specified number of components in the C&C model. The parent component for each new subcomponent is chosen randomly using the Java random number generator. A component is only added as a subcomponent if the parent has less than the maximal allowed number of subcomponents. Otherwise another parent is chosen at random. Similarly, the ports generated by the algorithm are randomly placed on the components in the C&C model while satisfying the constraint of the maximal number of ports per component. Initially ports do not have a type and a direction. The type and the direction of ports are set by the algorithm during the addition of connectors to the C&C model. This ensures that only ports of the same type and compatible directions are connected. The algorithm picks a port and randomly chooses the port's direction. It then computes all the possible targets of connectors in the C&C model and chooses one. The list of possible connector targets includes ports that are sources of other connectors but not those that are already targets of connectors (see well-formedness rules for connectors in C&C models in Definition 2.2, Item 8).

We further implemented a generator for random views. For a given C&C model, the views generator works in two steps. First, it clones the C&C model and eliminates some components, ports, and connectors based on the following parameters: number of components to keep, maximal number of ports to keep, and maximal number of connectors to keep (the actual number of ports and connectors in the view depends also on the number of ports and connectors left on the components to keep). Moreover, chains of concrete connectors are replaced by corresponding abstract connectors. Second, the algorithm randomly applies one or more of the mutations shown in Table 4.18.

Three of the seven mutation shown in Table 4.18 are benign. We call a mutation benign if it preserves satisfaction for all possible mutated elements in C&C views. The first three mutations introduce port types, component names, and port names that are not in the C&C model. Thus after any of these mutations has been applied to a derived C&C view the C&C model does not satisfy the C&C view. The mutation `SwitchCmpNames` might preserve satisfaction but not in all cases. It can change both satisfaction and non-satisfaction or have no effect if applied to two otherwise identical subcomponents of the same parent in the C&C model. The three latter mutations always preserve satisfaction since they only remove information from the C&C view.

Mutation	Effect	Benign
`ChangePortType`	changes the type of a randomly chosen port to a type not in the C&C model	no
`RenameCmp`	changes the name of a component to a new unique name not in the C&C model	no
`RenamePort`	changes the name of a port to a new unique name not in the C&C model	no
`SwitchCmpNames`	switches the names of two components	no
`RemovePortName`	removes the name of a randomly chosen port	yes
`RemovePortType`	removes the type of a randomly chosen port	yes
`RemoveAbsConnPorts`	removes either the source port or the target port information from an abstract connector	yes

Table 4.18.: Mutations for C&C views. A mutation is benign if it preserves satisfaction for all possible mutated elements in C&C views.

The benign mutations however do not necessarily preserve non-satisfaction. In case the single reasons for non-satisfaction of a view is a port that has no match in the C&C model, the `RemovePortName` mutation applied to this port will restore satisfaction.

In all cases, we implemented the mutations in a way that guarantees that the resulting mutated view is well-formed with respect to Definition 3.6. For example, if a component is renamed, the new name is also applied to the connectors that end or start at this component, i.e., the sets *Cmps* and *AbsCons* and the functions *subs* and *ports* are all updated accordingly. As another example, the mutation `RemovePortName` is only applied to a port if the port is has a known type (see Definition 3.6 for all well-formedness rules).

Experiments

We have set up two experiments to address the research question whether C&C views verification is feasible and scales for large C&C models.

Specifically, in the experiments we increased the number of components in the synthesized C&C models from 20 to 200, the maximal number of direct subcomponents per component was fixed to eight, the maximal number of port types was fixed to eight, the total number of ports in the C&C model was fixed to eight times the number of components in the C&C model, and the maximal number of connectors in the C&C model was set to half of the number of ports.

In our variable size setup we set the number of components in the view to be a fifth of the number of components in the C&C model, and set the number of mutations to a third of the number of components in the view.

In our fixed size setup we fixed the number of components in the view to 12 and the number of mutations to six. In both setups, we set the number of ports and abstract connectors to two times the number of components in the view. The first setup is designed to examine scalability. The fixed size setup where the views are of fixed size is designed to be more realistic, based on our experience with using C&C views verification, since views are typically of a smaller size not depending on the size of the C&C model.

In Figure 4.19 we show the results of two experiments of executing C&C views verification, including the computation of non-satisfaction reasons, on randomly generated C&C models and mutated views of different sizes.

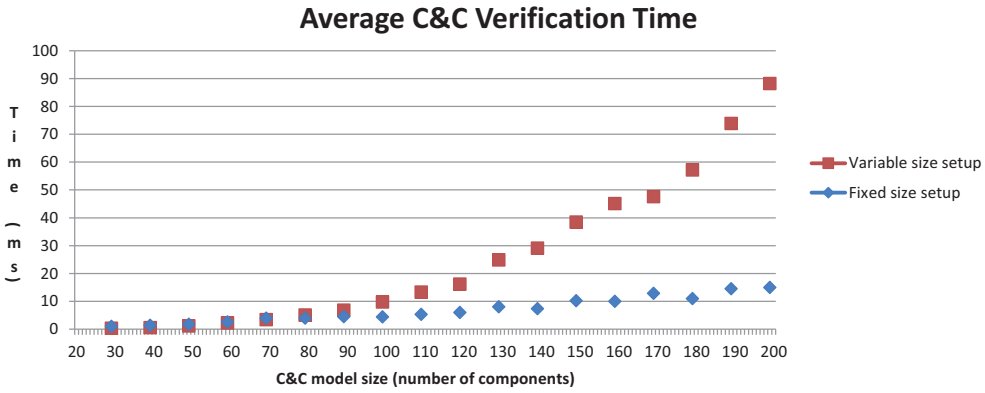


Figure 4.19.: Average times in milliseconds to decide satisfaction and compute the reasons for non-satisfaction, for the two setups, the variable size setup where the view size is a fifth of the C&C model size, and the fixed size setup where the view size is fixed to 12 components. Although the average times for the variable setup grow faster than the average times for the fixed setup, the absolute times recorded and the chart's growth clearly show that C&C views verification is feasible and scales well.

We executed all the experiments on an ordinary laptop computer, Intel Dual Core CPU, 2.8 GHz, running 64-bit Windows 7 and Java 1.7.0_17. We repeated the experiments 12 times for each C&C model size, from 20 to 200, for the two setups. Figure 4.19 reports the average times (in milliseconds) to decide satisfaction and compute the reasons for non-satisfaction, for the two setups. For every size from 20 to 200 — defining the number of components in the generated C&C model — we have computed the average times over the 12 independent runs with the random application of the mutations.

Although the average times for the variable setup grow faster than the average times for the fixed setup, the absolute times recorded and the chart's growth clearly show that C&C views verification is feasible and scales well. Moreover, in our experiments, average and maximal times to generate a witness were 11 ms and 595 ms respectively in the first setup, and 5 ms and 768 ms respectively in the second setup.

The running times obtained from the example systems in Table 4.17 seem to be in line with the times for the synthetic examples. For C&C models of sizes between 20 and 30 components (compared to 6-16 components in Table 4.17) the average running times for verification are 0.3ms (compared to 0.1ms) and for witness generation 1.8ms (compared to 1.8ms).

To conclude, the experiment results show that C&C views verification is feasible and scales well. Please note that fast and scalable performance comes at no surprise, since our algorithms are polynomial in the size of the input C&C model and views.

Threats to Validity

It is important to note that the experiment as designed only gives a rough indication of the algorithms' performance on real-world examples. On the one hand, the available C&C models of the example systems from various real-world and literature sources we reported on in Section 4.4.1 were too small to evaluate scalability. As demonstrated by the verification times shown in Table 4.17 these cases are handled almost instantly. On the other hand, the generated C&C models are not necessarily ones encountered in real-world systems. To address this bias we have made the C&C models generation parametrized and set the parameters to plausible values encountered in real-world C&C models, e.g., up to eight subcomponents per component. To give a closer estimate of the performance on real-world models these parameters should be validated on real-world examples.

To experiment with satisfied and non-satisfied views, we have created seven mutations that again are not necessarily the mutations resulting in non-satisfaction reasons encountered in real-world C&C models and views. We tried to address this issue by randomly choosing mutations and running the experiment multiple times. Thus, our results on the performance of the algorithm are only an indication for the performance on real-world examples.

4.4.3. Helpfulness of Generated Witnesses

We conducted a small user study to examine two high-level research questions: [RC1] is C&C verification difficult to do manually, and [RC2] are witnesses for satisfaction/non-satisfaction helpful. The study included a two-pages introduction on C&C views to read, 3 verification questions (each presenting one C&C view), and 3 questions about the usefulness of a set of witnesses that was presented to the user, all referring to a common C&C model. Two of the views in the first 3 questions had 2 non-satisfaction reasons each. The questions of each group were presented to the users in a random order to avoid a bias due to learning effect.

The study subjects were all CS graduate students or professional software engineers, all with some modeling background but no specific previous knowledge on our work on C&C views. No grades or other reward was involved. The study was anonymous and conducted online. We obtained complete set of answers from 17 subjects. The complete questionnaire and reference materials provided to the subjects are available in

Appendix H. The complete raw data including times for answering all questions and the comments of the subjects are available from [wwwu].

To answer [RC1] we measured the time spent on the first three questions, the correctness of the answers, and their completeness (identifying all reasons for non-satisfaction, where applicable). We also asked the subjects to report about their confidence in the correctness and completeness of their answers. The average (median) time to answer a verification question was 3.4 (2.9) minutes. 9 subjects (53%) identified non-satisfaction correctly and found all non-satisfaction reasons. 8 subjects (47%) missed at least one reason for non-satisfaction. Only 13 subjects (76%) identified satisfaction correctly (4 have 'identified' wrong non-satisfaction reasons). Only 3 subjects (18%) reported full confidence in the correctness of their answers and only 4 reported full confidence that they have identified all non-satisfaction reasons.

These results show that manual C&C verification is time-consuming and error prone. This justifies the need for automation.

To answer [RC2] we presented satisfaction results and witnesses (same C&C model as in the first three questions, but different views) to the subjects and asked them about the helpfulness of the witnesses. On average, 11.6 subjects (68%) reported to have found the witnesses we presented to them *helpful* or *very helpful* (top 2 out of 5 options). Only two subjects never found any of the witnesses helpful. 15 subjects found the witnesses helpful at least once.

These results are promising. Further investigation is required in order to identify which types of witnesses are more helpful than others, and how to improve witnesses helpfulness. Ideas for improvement we have received from the study subjects and other users include alternative witness constructions (e.g., not include the least common parent component), different witness presentation (e.g., using animation, or by visually overlaying the witness and the model to prevent the need for context switching), and richer textual explanations (e.g., adding text explicitly describing the elements shown in the witness).

Threats to Validity

Our set of 17 subjects is small and heterogeneous, e.g., in terms of previous modeling experience. The study involved a single small model (the pump station model) and only 3 verification questions and 3 witness usefulness questions based on views that we have designed. In the second part, we did not distinguish between different kinds of witnesses although their usefulness may vary. We also did not check for a correlation between the correctness of the answers in the first part to the perceived usefulness of witnesses in the second part. In the future we plan a larger study with more control on participants background and with more questions.

4.4.4. Lessons Learned

In our case studies running times for checking satisfaction, including parsing and witness generation for satisfaction or non-satisfaction, were in all cases very fast. For example,

running all the checks for the four example systems from Section 4.4.1 took less than 1 second, in total on an ordinary laptop computer. This seems to be in line with the results of our synthetic experiments of performance and scalability.

The number of witnesses computed for all C&C views in all our four example systems was at most 4. We believe that this is due to the relatively small number of design decisions typically documented in a single view. The underspecification mechanisms of C&C views allow to focus on only few elements per view to support comprehension. Another reason for the small number of witnesses is that by our definition, all checks for non-satisfaction reasons — except for missing components — ignore components in the view that are not shown in the C&C model.

On a more qualitative note, we observed that the generated natural language text was very helpful to us and to some colleagues we have presented our work to in understanding the reasons for non-satisfaction. Moreover, we found that witnesses with fewer elements make it easier for us to capture the reason for non-satisfaction. As a result, we have changed our initial witness generation for the interface mismatch case, from one that consists of the complete interface of the relevant component, to one that consists of only the port in question (as shown in Figure 4.8). We applied this change only to the port mismatch cases direction and type. It is not applicable to the missing port case since it requires showing all ports.

In a small user study we found out that manual verification of C&C models is a time-consuming and error prone task. Also, 15 out of 17 subjects found the generated witnesses helpful at least once for understanding positive and negative results of C&C views verification.

Finally, in almost all cases, we were able to express the structural properties we wanted to express using views and specifications. Only in one instance, in the robotic arm case study, we were not able to express a structural requirement, specifically, that no component should have the component `Sensor` and the component `Actuator` as direct subcomponents. Expressing this property requires quantification, which is not available in C&C views. As future work, one may suggest to define a structural C&C model specification language that supports quantification.

Additional experience and evaluation of our work with more case study C&C models and with the engineers that actually developed these C&C models will be valuable in further evaluating the usefulness and the contribution of our work to software architects.

4.5. Discussion

We now discuss several advanced topics and the strengths and limitations of our work.

4.5.1. Alternative Witnesses

One may consider generating witnesses for C&C views satisfaction and non-satisfaction that are different than the ones we currently generate.

One alternative relates to the witness we currently generate for a missing component. When the view includes a component that is not present in the C&C model, we currently generate a witness which is an empty view, annotated with a natural language description about the missing component. Instead, it may be valuable to generate a witness that shows the (minimal) subsystem of the C&C model in which the missing component should have been included. The minimal subsystem of a component c missing in the C&C model m is the most direct parent component of c in the view that also appears in m or the top component of m itself. This alternative may be considered a better fit with the characteristics of witnesses discussed in Section 4.3.2. In our current implementation we chose the empty view as a natural witness for a missing component. The algorithms in Procedure 7 may easily be adapted to generate alternative witnesses.

Further experience with witnesses for C&C views satisfaction and non-satisfaction is required in order to better evaluate the advantages and disadvantages of these alternatives.

4.5.2. Minimal Witnesses

The generated witnesses for non-satisfaction are minimal in terms of number of components, connectors, and ports, because there is only one possible witness to construct in each case, by definition. For the witness of satisfaction, minimality is more subtle, because, as we explain below, there may be more than one possible correct witness.

As explained in Section 4.3.3, our algorithm generates witnesses for satisfaction that are minimal with regard to the number of components and number of concrete chains of connectors that correspond to the abstract connectors that appear in the view. However, the witnesses we generate may not be minimal with regard to the total number of connectors. That is, if the C&C model includes several concrete chains of connectors that satisfy one abstract connector, our algorithm will add only one of these chains to the witness, but not necessarily the shortest one. Also, computing a shortest chain of connectors for each abstract connector does not guarantee a global minimum either, because in a minimal solution, some concrete connectors may potentially belong to multiple chains, i.e., be used to implement multiple abstract connectors.

Computing minimal witnesses for satisfaction is possible because the number of chains of connectors matching each abstract connector is finite and all combinations could be enumerated to find a global minimum. The computation is however more complex and computationally expensive. In the current implementation we chose a fast computation based on the heuristics described above rather than a slower computation of a global minimum.

4.5.3. Presentation Alternatives

The concrete presentation of witnesses may affect their effectiveness in explaining the reasons for satisfaction or non-satisfaction. We consider two kinds of presentations for the generated witnesses. First, a stand-alone representation consisting of the subset of the C&C model included in the witness. Second, alternatively, a highlighted representation

of the components and connectors of the witness on top of the concrete syntax of the C&C model document itself.

The advantage of the first representation is that it is typically small relative to the larger, complete C&C model, and thus easy to read and understand. The advantage of the second representation is that it does not require context switching; the view is displayed on top of the C&C model. However, this representation may not be suitable for large C&C models. Finally, the choice between the two presentations may also be related to whether we use a visual or a textual concrete syntax.

Our current implementation follows the first alternative.

4.5.4. Additional Abstraction Mechanisms

C&C views provide means for underspecification: not showing all components, not showing all ports and connectors, connecting ports or components using abstract connectors rather than chains of concrete connectors etc. Checking whether an C&C model satisfies a view may be viewed as checking whether the abstraction employed by the view is consistent with the C&C model.

One may consider applying abstraction mechanisms to the satisfaction problem, beyond the ones already embedded in C&C views. First, a *connectors* abstraction, where the satisfaction relation ignores all abstract and concrete connectors, and thus only checks for the existence of components, their ports, and their hierarchical configuration. Second, a *ports* abstraction, where the satisfaction relation ignores not only the connectors but also the ports. Finally, one may consider an even stronger abstraction, which ignores the hierarchical configuration.

Checking for satisfaction under these abstractions may be useful in some situations. Specifically, if the C&C model and view are very large, to the extent that it takes much time to check the former against the latter, one may use the abstractions suggested above as a faster, sound, but incomplete, approximation to satisfaction. For example, if an C&C model does not satisfy a view under the ports abstraction, i.e., when ignoring connectors and ports, then it clearly does not satisfy it without this abstraction. This means that the analysis with the abstraction is sound. On the other hand, if the C&C model satisfies a view under the ports abstraction, it may well be the case that it does not satisfy it without the abstraction because of missing connections. This means that the analysis with the abstraction is incomplete.

4.5.5. Identification of Components by Name

Our present work assumes a common name space for the C&C model and the view and considers an element's (component or port) name to represent its unique identity. In practice, it may be the case that different views use different names to denote the 'same' components or ports. A similar problem arises when components are renamed in the C&C model but not in existing views during the evolution of the modeled system. This will of course limit the value of the analysis.

Indeed, some works in the area of differencing and merging of models [XS07, KRPP09] use name matching based on lexical and structural similarity or other matching heuristics to find correspondence between two models. These techniques may be applied to component and connector architectures.

Thus, it may be possible to integrate a matching heuristics, based on lexical or structural similarity as a pre-processing step, before checking satisfaction. The input for the satisfaction algorithm will consist of the C&C model and the view after the different names that have been identified as referring to the same components or ports have been unified.

4.5.6. Verification of C&C Views Specification

The verification of a C&C model against a single C&C view can be extended to the verification of a C&C views specification as defined in Definition 3.9. Checking the satisfaction of a C&C views specification S by a C&C model m can be done by evaluating for every view $v \in V$ used in the specification whether $m \vDash v$. The Boolean results of the checks $m \vDash v$ then replace the view names in S as defined in Definition 3.10 of C&C views specification satisfaction. The remaining Boolean expression $S[v/(m \vDash v)]_{v \in V}$ can be evaluated in linear time.

The verification of a C&C model against a C&C views specifications thus requires a number of executions of the polynomial-time C&C views verification algorithm which is linear in the number of C&C views in the specification.

4.5.7. Applications of C&C Views Verification

We suggest example usage scenarios of C&C views in general in Section 3.2.1. One can use our work on C&C views verification to support maintaining a verified documentation of the C&C model using C&C views. Furthermore, our work supports an iterative 'check, debug, fix' cycle for C&C models and views. Given a set of views, as a specification, and a candidate C&C model representing the implementation, we check whether the model satisfies all the views. If it does not satisfy a view, we browse the generated witnesses and attempt to change the C&C model so that it satisfies the views. After the fix, we check again. We iterate until we find a C&C model that satisfies all the views.

Second, we may extend our views language with positive and negative modalities following similar modal extensions in other modeling languages [HM08, MRR11]. Positive views are ones that the C&C model should satisfy. Negative views are ones that the C&C model should not satisfy. Given a candidate version of the architecture, one can check it against all positive and negative views. The extension with modalities is an alternative to the views specifications as introduced in Section 3.5. While positive and negative modalities can also be expressed by views specifications, it might be more convenient for an engineer to directly add the modalities to the views.

4.6. Related Work

We discuss related work on various analyses related to C&C views, on structural abstraction mechanisms in ADLs, and on the relationships between structural and behavioral architectural views in the context of views verification. We have already discussed different usages of the term *view* in the modeling and software architecture literature in Section 3.7.5.

The key distinctive features of our work on C&C views verification are the focus on structure, the crosscutting 'by example' and partial characteristics of the views, the expressive power and formal nature of the specification approach, and the generation of witnesses to justify the results of the automated analysis.

4.6.1. Structural Abstraction and Verification in ADLs

A number of architecture description languages (ADLs) have been suggested in the literature (for a classification and survey see, e.g., [MT00] and [MDT07]). We discuss three widely used standard ADLs below. We repeat some of their structural specification mechanism mentioned in Section 3.7.5 and focus on their verification capabilities.

Armani and Acme

Armani [Mon98, Mon99] is a framework to define architectural styles and design rules for architectures. Armani is based on the ADL Acme [GMW00, GMW97]. An engineer can define styles and rules for systems using a constraint language based on first order predicate logic. Example predicates include $connected(c_1, c_2)$ and $reachable(c_1, c_2)$, which assert connectedness and transitive connectedness of components c_1 and c_2. Armani's constraints are evaluated over concrete architectures. The constraint language is integrated into AcmeStudio [wwwy] and constraints are automatically evaluated while editing architectures.

Acme's predicate language is in some ways more expressive than C&C views, as it has the flexibility of first order logic and allows quantification over components, connectors, ports, and roles. However, as far as we understand, the language neither supports the transitive subcomponent relation nor supports constructs to crosscut the bounds of the traditional implementation-based hierarchical decomposition of systems to their subsystems, types to subtypes etc.

Thus, while C&C views are less flexible than Acme's predicate language, it promotes a 'by example', easy to read and write specification style, and, significantly, provides natural means for abstraction, writing specifications that crosscut the traditional, implementation-oriented hierarchical decomposition of systems to subsystems, required to represent the concerns of interest of and the incomplete knowledge about the structure of the system available to different stakeholders involved in the software development process.

It may be possible to express the semantics of an C&C view as an Acme invariant. However, this invariant may be very long and difficult to understand relative to the

succinctness and intuitive nature of the view. Moreover, in case of non-satisfaction, C&C views verification provides a set of witnesses. We have not seen similar witnesses for non-satisfaction of Armani invariants in AcmeStudio.

More recently, Bhave et al. [BKGS11] have extended AcmeStudio to support structural consistency between heterogeneous models as architectural views, specifically for cyber-physical systems. View consistency is checked by verifying if a morphism exists between two typed graphs. The work mentions reporting back to the user so she is able to "spot the inconsistent elements" as future work. In addition, the work discusses a single case study and provides no performance and scalability results. We currently do not deal with heterogeneous models, but instead focus on structural properties and on the abstraction of direct containment and connectivity. Unlike this work, we do report on several example systems, and performance results that examine the usefulness of our solution.

AADL

AADL (Architecture Analysis Design Language) [wwwa, FGH06] is an architecture description language. AADL is standardized by the Society for Automotive Engineers (SAE). To a certain extent, the language includes under-specification mechanisms similar to the ones available in C&C views. For example, AADL supports specifications with incomplete information of port types and with abstract flows, which show the source and sink of flows but not their complete path through the system. Abstract flows are similar to the abstract connectors in C&C views. However, unlike the abstract connectors of C&C views, AADL flows refer to control flow rather than to data flow. In this sense C&C views may be seen as complementing AADL. We have not found any previous work on checking the structure of AADL architectures against specifications (made of some kind of views or by other means).

SysML

SysML [Obj12b, Wei07] is a general-purpose modeling language for systems engineering applications. The language is defined as an extension of a subset of the Unified Modeling Language (UML) [Obj12a], using UML's profiling mechanism. Previous work in our group [GHK+08b, GHK+08a] described the use of views in the context of product lines, with a focus on the automotive domain, using SysML's internal block diagrams. SysML's internal block diagrams provide under-specification mechanisms, for example, to specify abstract connectors similar to the abstract connectors of C&C views. However, the question of verifying the structure of a C&C model against a view is discussed neither in these works nor in any other SysML related work we have found.

Behjati et al. [BYN+11] have encoded the AADL as a profile on top of SysML. The main purpose of their work is to make commercial tools that support SysML available for creating AADL models and in turn benefit from the analysis and modeling features of AADL. This integration of AADL and SysML may be a good starting point for an integration of C&C views with both languages and with standard, available tools.

4.6.2. Structural and Behavioral Architectural Views

Our present work on C&C views is intentionally limited to a structural viewpoint. Currently, behavior is abstracted away. Most ADLs, however, combine structure and behavior.

Many behavioral specification languages exist and some of them have related views. For example, linear temporal logic (LTL) [MP92, Pnu77] is a behavioral specification language, and scenarios, expressed, e.g., using live sequence charts (LSC) [DH01, HM08], may be considered as related behavioral views. As another example, in SysML, behavior is specified using activity diagrams and state machine diagrams.

In the behavioral case, a system's behavior is typically modeled using a state machine, and the behavioral properties this state machine needs to satisfy are expressed using LTL formulas or scenarios. Model-checking techniques [BK08] can be used to check whether a state machine satisfies a behavioral property and provide a counterexample in case of a negative answer [CGP99]. In the structural case, which is the focus of our work, the structure of a system is described using a C&C model and the properties it needs to satisfy are expressed using C&C views. The algorithm presented in Section 4.3 is used to check whether a C&C model satisfies a view. The witnesses we generate in case of non-satisfaction may be considered as the structural analogue to counterexamples in behavioral model checking; as in the behavioral case, where counterexamples may themselves be viewed as scenarios (at least in the context of LTL model checking). In the structural case we focus on with C&C views verification, witnesses for non-satisfaction are themselves views.

An interesting and challenging possible direction for future work is to combine the structural and behavioral viewpoints into a single views language, with a related formal verification technique.

To conclude this section, as the discussions above show, to the best of our knowledge, no previous work is directly comparable to C&C views verification and witness generation for the structure of C&C architectures.

Chapter 5.

Component and Connector Model Synthesis from Views Specifications

C&C views are partial models of a logical software architecture. They represent the concerns of interest of and the incomplete knowledge available to different stakeholders involved in a software development process. As such, they may be related to the participants in a specific use case or scenario and thus typically crosscut the traditional hierarchical, implementation-oriented decomposition of systems to subsystems.

A system's C&C architecture is typically complex; it is not designed by a single architect and is not completely described in a single document. Moreover, some C&C models may be bound to reuse library or third-party components designed and documented elsewhere. Thus, we consider a setup where many different, incomplete, relatively small views of the C&C model are provided by architects responsible for subsystems, for the implementation of specific features, use cases, or functionality, which crosscut the boundaries of components. Such views may be developed by separate, distributed teams, each focusing on only some aspects of the system and its development and having only partial knowledge of the system as a whole. Moreover, a team may have several, alternative solutions that address the same concern, and some knowledge about designs that must not be used. To move forward in the development process and to enable implementation, these partial views and their design rationales should be integrated and then realized into a single, complete C&C model of the software architecture. However, such an integration is a complex and challenging task.

In this chapter we introduce the C&C model synthesis problem: constructing a C&C model from a declarative specification made of C&C views, representing mandatory, alternative, and negative structural properties. The modalities of the C&C views are specified in C&C views specifications as introduced in Section 3.5.

The C&C views synthesis problem is NP-hard. We solve it, in a bounded scope, using a reduction to SAT, via Alloy [Jac06] (since the problem is NP-hard, the use of Alloy/SAT to solve it is justified). If a satisfying assignment is found by Alloy, we translate it back into a complete C&C model and present it to the architect. Our solution for synthesizing C&C models is extensible. We demonstrate how the solution supports variability in the views language and provide support for library components.

We have reported on parts of this work in [MRR13].

Chapter outline and contributions

Section 5.1 illustrates an example for synthesizing a C&C model from multiple C&C views. We define the C&C views synthesis problem in Section 5.2 and show that the problem is NP-hard. Our main contribution of this chapter is an extensible formalization of the C&C views synthesis problem in Alloy presented in Section 5.3. Extensions of the C&C views synthesis for handling new features of the C&C views language are defined in Section 5.4. Section 5.5 presents extensions for synthesizing C&C models that conform to architectural styles.

We present our implementation of C&C views synthesis and an evaluation based on applying it to four example systems in Section 5.6. Section 5.7 discusses strengths and weaknesses of our solution and implementation for solving the synthesis problem. We examine related work in Section 5.8.

5.1. Component and Connector Model Synthesis Example

We present an overview of C&C views synthesis using an example, adopted from an industrial model of a robotic arm[1], which is typical to a cyber-physical or an embedded system. We focus here on a single joint of this arm.

5.1.1. Synthesizing a C&C Model

Imagine a scenario where a team of engineers is developing the C&C model of the robotic arm based on C&C views documenting the partial knowledge available and the design decisions made by the team.

One of the engineers creates the view `RJFunction` shown in Figure 5.1 (a) to describe the system's C&C model from the point of view of the team responsible for its function: the `RotationalJoint` contains the component `Cylinder` and the component `Sensor` that is connected to the component `Actuator`. As a C&C view, `RJFunction` is partial, so it may not contain all components of the system. Moreover, while the components shown inside the joint must actually be inside the joint, they may be nested within some of its subcomponents (not shown in this view). On the other hand, the view specifies that the three subcomponents, `Cylinder`, `Sensor`, and `Actuator` are not nested within one another. Finally, the component `Sensor` and the component `Actuator` must be connected, but their connection is not necessarily direct.

The senior architect initially created the view `RJStructure` shown in Figure 5.1 (b) to provide a high-level description of the `RotationalJoint` structure, some of the components it contains and the connections between them. In this C&C view the architect describes her knowledge of the joint. The view shows the name `angle` and type `float` of an incoming port of the component `Cylinder` for a connection (not necessarily direct) coming from the component `Body`. The architect decided to document the

[1]We thank Ali Muhammad, Remote Operation and Virtual Reality Group, VTT Tampere, Finland, for allowing us to use this model.

detailed information of the port of the component `Cylinder`, while she laid less focus on other connections, e.g., from the component `ServoValve` to the component `Body`.

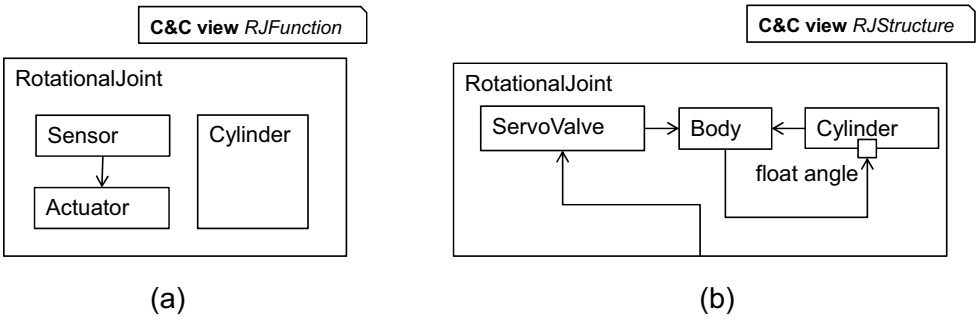


Figure 5.1.: The C&C views `RJFunction` and `RJStructure` documenting partial knowledge available to the engineers. Please note that the implementation details about the connection between the components `Sensor` and `Actuator` are not given in the view `RJFunction`, e.g., the source and target ports do not appear in the view.

When specifying the structure of the component `Body` the team had a long discussion about the placement of the component `Sensor`. The C&C views `BodySensorIn` and `BodySensorOut` shown in Figure 5.2 (a) and (b) describe two alternatives for the decomposition of the component `Body` with focus on its internal structure. The first specifies four subcomponents of the component `Body` (not necessarily direct subcomponents, not necessarily all of them) and the connections between them (again, not necessarily all connections, not necessarily direct ones). The second view documents the alternative, where the component `Sensor` is outside the component `Body`.

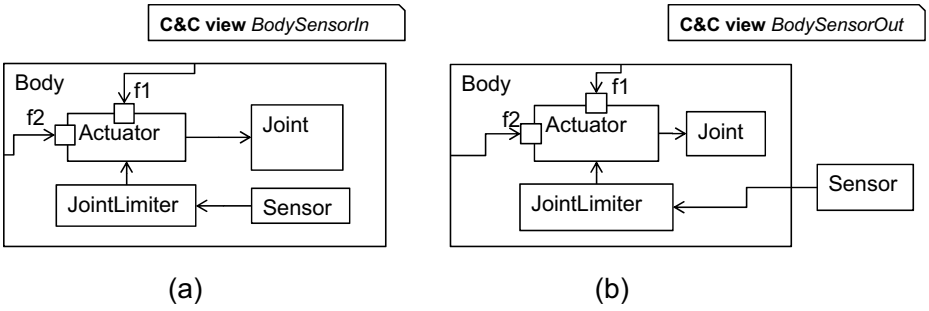


Figure 5.2.: The C&C views `BodySensorIn` and `BodySensorOut` showing two alternative designs with different placements of the component `Sensor`.

The team's expert for sensors added the C&C view `SensorConnections` shown in

Figure 5.3 (a) to document her knowledge about the sensor component and its outgoing ports. The view shows that the component `Sensor` is connected to the component `Cylinder` via an output port named `val2` of type `int` and to a component named `JointLimiter` via the output port `val1` of type `float`. Again, the C&C view is partial, thus in the complete C&C model the connections shown may be indirect and `Sensor` may be connected to additional components.

Finally, the sensor expert also added the C&C view `ASDependence` shown in Figure 5.3 (b), which shows the component `Actuator` and the component `Sensor` inside the component `Body`. It describes some domain knowledge concerning a requirement for independence between the component `Actuator` and the component `Sensor`. Thus, it is used in the specification (see below) in a negated form, to not allow a C&C model where the component `Actuator` and the component `Sensor` are both inside the same component, in this case, `Body`.

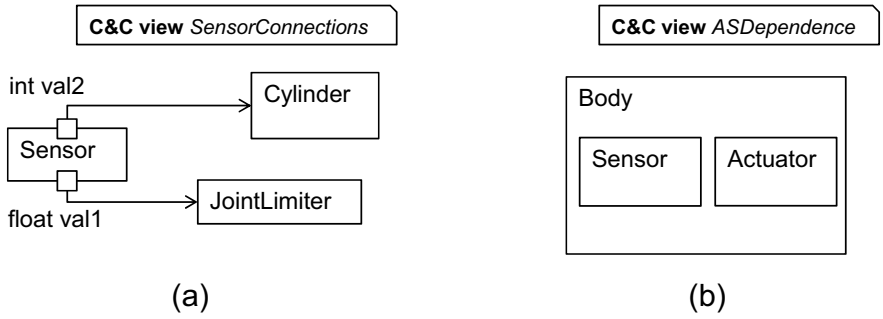


Figure 5.3.: The C&C view `SensorConnections` documents details about the connections starting from the component `Sensor`. The C&C view `ASDependence` depicts a design where the components `Sensor` and `Actuator` are both contained in the component `Body`.

After all C&C views have been collected from the team members the senior architect compiles the views specification S_1 to capture the views about the function and structure of the joint, the alternative designs regarding sensor placement, the details about the sensor connections, and the forbidden design where the components `Sensor` and `Actuator` are both contained inside the component `Body`:

$$S_1 = \texttt{RJFunction} \wedge \texttt{RJStructure} \wedge (\texttt{BodySensorIn} \vee \texttt{BodySensorOut}) \wedge \texttt{SensorConnections} \wedge \neg\texttt{ASDependence}.$$

Is there a C&C model that satisfies this specification? Our work provides a fully automated and constructive answer to this question. Specifically, given this specification, our tool provides a positive answer and outputs the C&C model shown in Figure 5.4. For readability, we omit some port names, types, and internal connectors from the diagram.

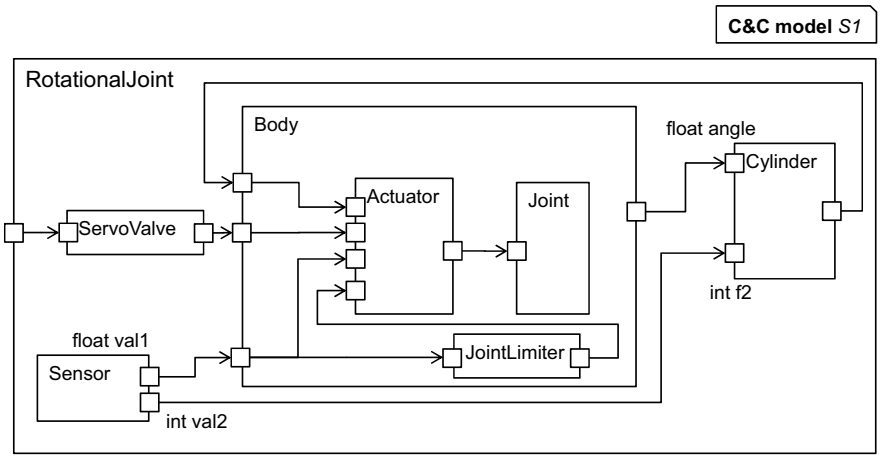


Figure 5.4.: A C&C model with 20 ports satisfying the C&C views specification S_1.

5.1.2. An Unsatisfiable Specification

The member of another engineering team suggests to add a C&C view used in a previous project. The additional view `OldDesign` is shown in Figure 5.5. The view specifies that the component `Actuator` is connected to the component `Cylinder` and that both components are contained inside the component `Body` (although not necessarily directly). It also depicts the name `angle` and type `int` of the component `Cylinder`'s incoming port for a connection (not necessarily direct) coming from the component `Actuator`.

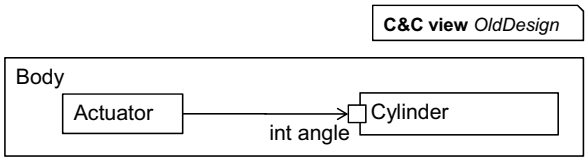


Figure 5.5.: The C&C view `OldDesign` describes the relation between components `Actuator` and `Cylinder`.

Is there a C&C model that satisfies the revised specification S_2 (consisting of S_1 after adding `OldDesign` as another conjunct)?

$$S_2 = \texttt{RJFunction} \wedge \texttt{RJStructure} \wedge (\texttt{BodySensorIn} \vee \texttt{BodySensorOut}) \wedge \texttt{SensorConnections} \wedge \neg\texttt{ASDependence} \wedge \texttt{OldDesign}$$

Our tool identifies that S_2 is unsatisfiable and informs that no C&C model exists that satisfies S_2. One reason relates to the containment relation between the component `Body` and the component `Cylinder`: according to the view `RJStructure`, the two components are not contained within one another; according to the view `OldDesign`, the latter is contained within the former. Another reason is the type conflict `float` vs. `int` for the component `Cylinder`'s incoming port `angle`.

5.2. Synthesis Problem Definition

The C&C model synthesis problem for a C&C views specification is defined in Definition 5.6. The input for the synthesis problem is a set of C&C views (defined in Definition 3.6) and a views specification as defined in Section 3.5. The output of the synthesis problem is a C&C model that satisfies the views specification, if one exists.

Definition 5.6 (C&C views synthesis problem)**.** Given a C&C views specification S over a set of views *views*, find a C&C model m such that $m \vDash S$ if such a model exists. △

5.2.1. Synthesis from Views Specifications is NP-hard

We show that the C&C views synthesis problem for C&C views (even without connectors) is NP-hard, using a reduction from 3SAT [GJ79]. An instance of the 3SAT problem is a propositional formula given in conjunctive normal form (conjunction over clauses of disjunction) with at most three literals per clause. A literal is the possibly negated name of a Boolean variable. The decision problem is to decide whether there exists an assignment of the Boolean variables that satisfies the formula. This problem is known to be NP-complete [GJ79]. If an NP-complete problem can be reduced by a polynomial transformation to another problem the latter is NP-hard.

The main idea behind our reduction of 3SAT to the C&C views synthesis problem is to translate each Boolean variable into two views for representing positive and negative evaluations of the variable as shown in Figure 5.7. No C&C model can satisfy both views since they contain a contradiction in the hierarchy of the two components $\texttt{xT}_\texttt{i}$ and $\texttt{xF}_\texttt{i}$ contained in the views, i.e., $\nexists m : m \vDash vT_i \wedge m \vDash vT_i$. We then express the 3SAT problem in terms of these views as a C&C views synthesis problem.

Lemma 5.8. The C&C views synthesis problem defined in Definition 5.6 is NP-hard.

Proof. We show that the C&C views synthesis problem is NP-hard by a reduction of the 3SAT problem to the C&C views synthesis problem.

Given a 3SAT formula over variables $x_1, \ldots, x_n$ in clauses $c_1, \ldots, c_m$ we construct the following C&C views specification. First, for each variable x_i we define two components $\texttt{xT}_\texttt{i}$ and $\texttt{xF}_\texttt{i}$ and two views vT_i and vF_i such that in vT_i, the component $\texttt{xT}_\texttt{i}$ contains the component $\texttt{xF}_\texttt{i}$, and in vF_i, the component $\texttt{xF}_\texttt{i}$ contains the component $\texttt{xT}_\texttt{i}$ (see Figure 5.7). Intuitively, vT_i (vF_i) represents a positive (negative) valuation for x_i. Obviously, a given C&C model can only satisfy one of them. Second, for each clause c_j we create a views clause cV_j that includes a disjunction of up to three views, vT_i or vF_i for

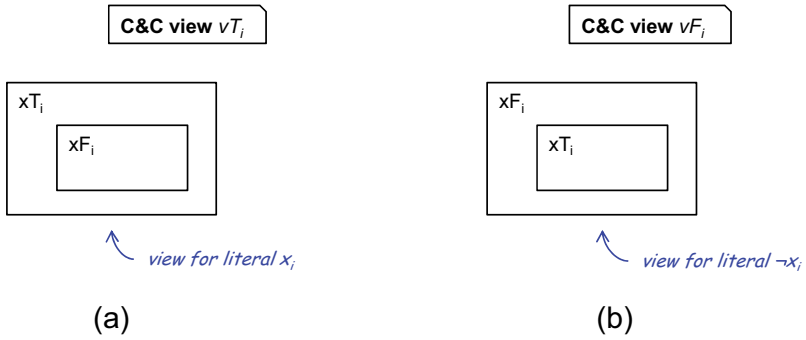


Figure 5.7.: Translation of every variable x_i to views vT_i and vF_i for representing the positive and negative evaluations of variables.

each variable x_i in c_j: if x_i appears positive in c_j we use vT_i, if it appears negated we use vF_i.

We create a propositional formula for the C&C views specification consisting of a conjunction of the views disjunction $vT_i \vee vF_i$ for all $1 \leq i \leq n$, and the views clauses cV_j for all $1 \leq j \leq m$:

$$\bigwedge_{i=1}^{n}(vT_i \vee vF_i) \wedge \bigwedge_{j=1}^{m} cV_j \tag{5.1}$$

The first part makes sure that any satisfying C&C model encodes a valuation for every variable of the 3SAT formula. The second part encodes the satisfaction constraints of the 3SAT formula on its variables. Thus, the 3SAT formula has an assignment if and only if the specification has a satisfying C&C model.

The generation of the first part of expression (5.1) is linear in the number of variables $x_1, \ldots, x_n$ of the 3SAT problem. The generation of the second part of expression (5.1) is linear in the number of clauses $c_1, \ldots, c_m$ of the 3SAT problem.

The linear reduction of the NP-hard problem 3SAT to the C&C views synthesis problem shows that C&C views synthesis is NP-hard. □

In Chapter 4 we have presented a verification algorithm with polynomial time complexity. It is important to note that this algorithm can verify a given C&C model against a C&C views specification in polynomial time as discussed in Section 4.5.6 while synthesizing a satisfying C&C model is NP-hard.

We present a solution to the synthesis problem by a reduction to a SAT solver via the Alloy Analyzer. Our experiments show that despite the non-polynomial complexity, we can synthesize C&C models with 5-15 components from 5-10 views in under a minute on an ordinary desktop computer.

5.3. Component and Connector Model Synthesis

Our solution for solving C&C model synthesis is based on a reduction of the synthesis problem to a model finding problem for the Alloy Analyzer. The Alloy Analyzer is able to analyze modules written in the Alloy language based on first order relational logic. Alloy is a model finder that is complete only in a bounded scope. Thus, our solution to C&C views synthesis is also restricted to a user defined upper bound of the numbers of elements in the C&C model. Due to the nature of our solution a user only needs to define an upper bound for the number of ports and the number of port names (see Section 5.7.1).

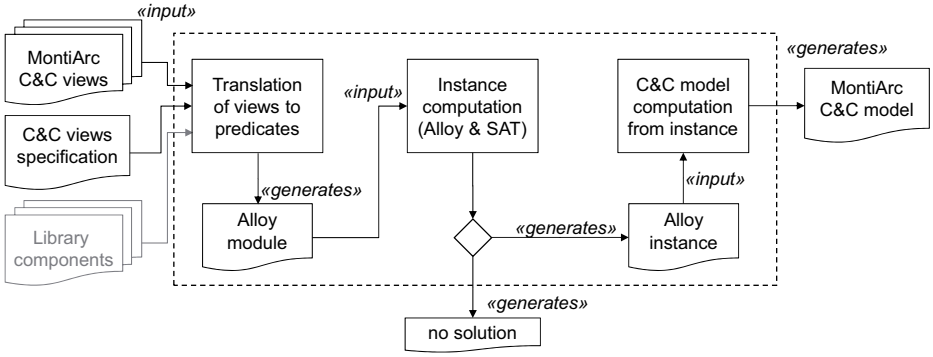


Figure 5.9.: An illustration of our approach to C&C model synthesis. The rectangular boxes represent the main computation steps. The translation of C&C views into predicates is described in Section 5.3.4. The translation of Alloy instances into C&C models is described in Section 5.3.6. The optional input of library components is described in Section 5.4.2.

Our approach is illustrated in Figure 5.9. The input to the synthesis engine is a set of views, the views specification, and optional definitions of library components. We translate each view into a predicate over a C&C model domain expressed in Alloy. The views specification is translated into an expression combining the generated view predicates into an integrated Alloy module. We then run the Alloy Analyzer on the generated module. If an instance is found, we translate this instance into an abstract syntax tree using MontiArc's APIs and use the pretty printer generated by MontiCore to print the MontiArc model of a C&C model satisfying the views specification.

The formulation of the C&C views synthesis problem in Alloy consists of four parts: a fixed set of signatures and facts describing a metamodel for C&C models inside Alloy (see Section 5.3.2), a fixed set of predicates used as a language to specify the semantics of C&C views as predicates over satisfying C&C models (see Section 5.3.3), a set of signatures and predicates derived from the specific input views (see Section 5.3.4), and the specification's propositional formula (see Section 5.3.5).

5.3.1. Alloy and the Alloy Analyzer

Alloy refers to both, a textual specification language to define Alloy modules [Jac06] and the Alloy Analyzer [wwwb], a tool that offers analysis support for these modules.

The Alloy specification language is based on relational first-order logic. The basic elements of this logic are sets and relations. An Alloy module consists of signature declarations, predicates, functions, facts, and commands. A signature declaration may contain the signature's cardinality and define generalization or subset relations to other signatures. As an example, consider the Alloy module `alloyExample` in Listing 5.10. The singleton signatures `IN` and `OUT` (ll. 4-5) extend the abstract signature `Direction` (l. 3).

Alloy

```
1  module alloyExample
2
3  abstract sig Direction {}
4  one sig IN extends Direction{}
5  one sig OUT extends Direction{}
6
7  sig Name {}
8
9  sig Port {
10   name: one Name,
11   dir: one Direction
12 }
13
14 fact uniquePortNames {
15   all p1, p2 : Port |
16     p1.name = p2.name implies p1 = p2
17 }
18
19 run {} for 5 but exactly 3 Port
```

Listing 5.10: A simple Alloy example of Ports with unique names and a direction.

Signature declarations may also contain fields with multiplicities referring to other signatures. The signature `Port` has a field `name` whose value is exactly one instance of the signature `Name` (see l. 10, Listing 5.10). Each signature denotes a set of instances called atoms. Each field represents a relation between the atoms of its parent signature and the atoms of the field's type. Relations are thus sets of tuples of atoms.

Alloy uses facts to constrain the possible valuations of signatures and relations. The fact `uniquePortNames` (ll. 14-17) quantifies over all port atoms `p1` and `p2` of the signature `Port` (l. 15). The body of the quantification states that equal names of port `p1` and `p2` imply that the ports are identical (l. 16). The notation `p1.name` describes the left relational join of the relation `name` $\subseteq$ `Port` $\times$ `Name` with the singleton relation {`p1`} $\subseteq$ `Port`, which in this case yields a single atom from the set denoted by the

signature `Name`.

Predicates are parametrized constraints, which can be included in other predicates or facts. Functions may have parameters and return atoms or sets of atoms. A complete language reference is given in [Jac06][2].

The Alloy Analyzer is a fully automated constraint solver for Alloy modules. To perform analysis for finding models that satisfy or violate predicates the search space has to be bounded by the user. This bound is set for every signature and thus also implies bounds on all relations. In the example in Listing 5.10 the run command in line 19 bounds the number of names to any number from 0 to 5 and the number of ports to exactly 3.

The Alloy Analyzer translates the bounded relational problem into a propositional formula and solves it using a SAT solver [GKSS08]. This approach has the advantage, that problems formulated in first order relational logic which is in general undecidable [HR04] can be solved in a bounded scope where the problems are decidable. A disadvantage is the limitation to a user specified bound (analysis scope). If the bound on the scope is not known and the Alloy analyzer does not find a solution, it is unclear whether one exists in a larger scope. The search within a scope is complete, i.e., if a solution in the scope exists, it is found.

5.3.2. C&C Metamodel in Alloy

Our approach to solve the synthesis problem is to define the domain of valid C&C models as an Alloy metamodel using signatures with fields (Listing 5.11) and facts (Listing 5.12 and Listing 5.13). This metamodel is generic and independent of the specific synthesis problem. A specific synthesis problem is then an additional set of constraints that needs to be solved by computing a valid model with respect to the constraints and the metamodel.

Listing 5.11 shows the Alloy signatures describing the metamodel for C&C models, e.g., the signatures `Component` (ll. 1-5) and `Port` (ll. 15-22). A component has a set of ports, a set of sub components, and at most one parent. The `parent` value of a component is derived from the `subComponents` relation using the fact `subComponentsAndParents` shown in Listing 5.12, ll. 1-4 . Ports in the metamodel have a direction (`IN` or `OUT`), a type and a name. Connectors in C&C models are modeled as the field `receivingPorts` of the signature `Port` (Listing 5.11, l. 19). The relation `receivingPorts` relates each port to its set of immediately connected ports. Analogously the field `sendingPort` (l. 21) relates a port to the single direct sender port, if one exists.

The signature and field definitions, as depicted in Listing 5.11, allow the instantiation of structures that represent invalid C&C models. We thus define a set of facts about legal instantiations of the signatures and relations. Listing 5.12 shows facts about the `Component` signature, its `parent` relation, and its `subComponents` relation. The

[2]The language reference is also available from `http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf` (accessed 04/2013)

Alloy

```
abstract sig Component {
  ports : set Port,
  subComponents : set Component,
  parent : lone Component // fixed by fact subComponentsAndParents
}

abstract sig Direction {}
one sig IN extends Direction{}
one sig OUT extends Direction{}

sig PortName {}

sig Type {}

sig Port {
  direction: one Direction,
  type : one Type,
  name: one PortName,
  receivingPorts : set Port,
  owner : one Component, // fixed by fact portsAndOwners
  sendingPort : lone Port // fixed by fact portsAndSender
}
```

Listing 5.11: The Alloy signatures of the C&C model metamodel in Alloy.

Alloy

```
fact subComponentsAndParents {
  all ch, par : Component |
    (ch in par.subComponents iff ch.parent = par)
}

fact subComponentsAcyclic {
  no comp : Component |
    comp in comp.^subComponents
}
```

Listing 5.12: Alloy facts about components in the C&C model metamodel in Alloy.

fact `subComponentsAndParents` determines the single parent component of a child component (see the well-formedness rule in Definition 2.2, Item 6). The fact `subComponentsAcyclic` states that no component is contained in the transitive closure of its subcomponent relation.

Listing 5.13 contains various facts about the ports of components and the C&C model's connectors expressed as relations between ports. The fact `portsAndOwners` constrains the field `owner` of ports based on the values of the field `ports` of a component. In a

Alloy

```
1  fact portsAndOwners {
2    all cmp : Component | all port: cmp.ports |
3      cmp = port.owner
4  }
5
6  fact allPortsOwned {
7    Port in Component.ports
8  }
9
10 fact portsOfComponentHaveUniqueNames {
11   all c: Component | all disj p1, p2: c.ports |
12     (p1.name != p2.name)
13 }
```

Listing 5.13: Alloy facts about ports in the C&C model metamodel in Alloy.

legal Alloy instance representing a C&C model the ports owned by a component (relation `ports`) are thus the ones that have the component as their owner (relation `owner`).

The fact `allPortsOwned` (ll. 6-8) states that all atoms of the signature `Port` are contained in the join of the signature `Component` with the relation `ports`, i.e., all instantiated ports belong to at least one component. Since we model connectors in C&C models using a relation between ports, it is important to not have connections from dangling ports.

The fact `portsOfComponentHaveUniqueNames` (ll. 10-13) ensures that no two ports that belong to the same component have the same name (see the well-formedness rule in Definition 2.2, Item 7).

Listing 5.14 contains facts about the relations of the signature `Port` that we use to represent connectors in C&C models. The first fact `portsAndSender` defines the relation `sendingPort` in terms of the relation `receivingPorts` to ensure consistency of the relations. The second fact `notConnectedToSelf` states that a port is never its own sender and never among its own receiving ports.

The fact `portsConnectedLegally` (ll. 13-29) states that a sending port and all its receiving ports have the same type (l. 15) and that each pair belongs to one of the four possible cases for connectors in C&C models. The cases are: a direct or pass-through connection from an input to an output port (l. 17), a connector from a parent component to an input port of one of its subcomponents (l. 20), a connector from a subcomponent to an output port of the parent component (l. 23), and a connector from a subcomponent to another subcomponent. These four cases correspond to the four cases listed in the definition of C&C models (see Definition 2.2, Item 9).

Alloy

```
1  fact portsAndSender {
2    all disj sender, receiver: Port |
3      (sender = receiver.sendingPort iff
4        receiver in sender.receivingPorts)
5  }
6
7  fact notConnectedToSelf {
8    all p: Port |
9      p != p.sendingPort and
10     p not in p.receivingPorts
11 }
12
13 fact portsConnectedLegally {
14   all sender: Port | all receiver: (sender.receivingPorts-sender) |
15     (receiver.type = sender.type and
16       // direct connector
17       ( (receiver.owner = sender.owner and sender.direction = IN
18                                   and receiver.direction= OUT) or
19       // toChildConnector
20       (receiver.owner in sender.owner.subComponents and
21               sender.direction = IN and receiver.direction= IN) or
22       // fromChildConnector
23       (sender.owner in receiver.owner.subComponents and
24             sender.direction = OUT and receiver.direction= OUT) or
25       // subComponentConnector
26       (receiver.owner != sender.owner and
27        sender.owner.parent = receiver.owner.parent and
28        sender.direction = OUT and receiver.direction= IN) ) )
29 }
```

Listing 5.14: Alloy facts about the representation of connectors in the C&C model metamodel in Alloy.

5.3.3. C&C Views Semantics in Alloy

We define the semantics of each view by a translation of the view into an Alloy predicate over valid C&C models of the metamodel described in Section 5.3.2. The predicate for a view is composed of parametrized predicates about the relations of components, component connectivity and component interfaces. We give an overview of these basic predicates in Listings 5.15, 5.16, and 5.17.

Listing 5.15 shows the predicate `contains` parametrized with a component `parent` and a component `child`. The predicate holds if and only if the component `child` is in the transitive closure of the subcomponent relation of the component `parent`. The complementary predicate `independentSet` (ll. 5-9) defines the semantics of a set of components where no two components contain each other.

Alloy

```
pred contains [parent: Component, child: Component] {
  child in parent.^(subComponents)
}

pred independentSet [components: set Component] {
  all disj c1, c2 : components |
    ( (no c1.subComponents)  or
      (not contains[c1, c2]) )
}
```

Listing 5.15: Alloy predicates about the containment relation of components to define the semantics of C&C views.

Alloy

```
pred connected[sender: Component, receiver: Component] {
  some p : receiver.ports |
    p in sender.ports.^receivingPorts
}

pred connectedWithPortNames[sender: Component, sendName : PortName,
                            receiver: Component, recvName: PortName] {
  some sp : sender.ports | some rp : receiver.ports |
    rp.name = recvName and
    rp in sp.^receivingPorts and
    sp.name = sendName and
    sp in rp.^~receivingPorts
}

pred connectedWithReceiverPortName[sender: Component,
                           receiver: Component, recvName : PortName] {
  some sendName : sender.ports.name |
    connectedWithPortNames[sender, sendName, receiver, recvName]
}

pred connectedWithSenderPortName[sender: Component,
                           sendName : PortName, receiver: Component] {
  some recvName : receiver.ports.name |
    connectedWithPortNames[sender, sendName, receiver, recvName]
}
```

Listing 5.16: Alloy predicates about the connectedness of components.

Listing 5.16 contains predicates about the connectedness of components in a view. The first predicate `connected` is parametrized with the components `sender` and `receiver`. The predicate holds if and only if component `receiver` has a port contained

in the transitive closure of the receivingPorts relation of sender's ports. This predicate expresses the semantics of an abstract connector with source sender and target receiver and no information on the source or target ports.

The predicate connectedWithPortNames (ll. 6-13) is instantiated to express the semantics of an abstract connector for which the source component and source port as well as the target component and target port are known. The similar predicate connectedWithReceiverPortName (ll. 15-19) handles the case where the target port is known but not the source port. This predicate is defined by stating that there is a port name sendName in the set of names of the set of ports of the source component sender that satisfies the predicate connectedWithPortNames. Analogously, the predicate connectedWithSenderPortName (ll. 21-25) handles the case where the source port is given but not the target port.

Alloy

```
1  pred untypedPort[cmp: Component, dir: Direction,
2                                                  portName: PortName] {
3    some port : cmp.ports |
4      port.direction = dir and
5      port.name = portName
6  }
7
8  pred unnamedPort[cmp: Component, dir: Direction, pType: Type] {
9    some port : cmp.ports |
10     port.direction = dir and
11     port.type = pType
12 }
13
14 pred portOfComponent[cmp: Component, dir: Direction, pType: Type,
15                                                 portName: PortName] {
16   some port : cmp.ports |
17     port.direction = dir and
18     port.name = portName and
19     port.type = pType
20 }
```

Listing 5.17: Alloy predicates about the interfaces of components to define the semantics of C&C views.

Listing 5.17 contains predicates to formulate the semantics of component interfaces (ports of the component). The first parametrized predicate untypedPort states that the component cmp has a port with the direction dir, and the port name portName. The predicate unnamedPort (ll. 8-12) is analogously defined for ports with unknown name but known type. The last predicate in Listing 5.17 can be instantiated with a component, port direction, port type, and port name to require a port with the given properties in every satisfying C&C model.

5.3.4. C&C Views Translation

The translation of views into Alloy predicates has two stages. First, we collect from the input views the component names, port names, and types to define signatures extending `Component`, `PortName`, and `Type`. These elements are collected from all views and serve as the building blocks for instances representing valid C&C models. Second, we translate each view into a predicate, which expresses the view's semantics using the predicates defined in Section 5.3.3.

An overview of the translation is given in Figure 5.18 using a formal notation for translation rules defined in Appendix B. The rules operate on the abstract syntax of C&C views, e.g., the line $\forall view \in views$ executes the following indented lines for each C&C view *view* in the set *views*. The result produced in the target language (here the Alloy language) is marked by `underlining the elements` of the target language syntax. Explanations of the execution of the rules are given in Appendix B. We provide concrete examples in the figures of most of the rules referenced in Figure 5.18, e.g., for rule V2 in Figure 5.20. A complete example for the translation of a C&C views specification including all views is presented in Appendix I.

Translation Rule

Overview of the translation of a set of C&C views *views* into an Alloy module:

```
module CnCViewsSynthesis

// generic part: signatures, facts, predicates
```
see Section 5.3.2 and Section 5.3.3

executeRule(V1) *see Figure 5.19 for body of rule V1*
```
// all components used in views
```
executeRule(V2) *see Figure 5.20 for body of rule V2*
```
// all port names used in views
```
executeRule(V3) *see Figure 5.21 for body of rule V3*
```
// all types used in views
```
executeRule(V4) *see Figure 5.22 for body of rule V4*

∀*view* ∈ *views* :
```
pred view.name {
   // all components shown in the view exist
```
executeRule(P1) *see Figure 5.24 for body of rule P1*
```
   // ensure all independent sets
```
executeRule(P2) *see Figure 5.25 for body of rule P2*
```
   // component containment relation
```
executeRule(P3) *see Figure 5.26 for body of rule P3*
```
   // ports of components
```
executeRule(P4) *see Figure 5.27 for body of rule P4*
```
   // connections between components
```
executeRule(P5) *see Figure 5.28 for body of rule P5*
```
}

// synthesis predicate
```
see Section 5.3.5

```
// synthesis run command
```
see Section 5.3.5

Figure 5.18.: Overview of the translation of a C&C views specification into an Alloy module. Comments above the rule executions belong to the respective rules. The syntactical elements of the translation rules are listed in Appendix B.

The first rule V1 shown in Figure 5.19 states as a fact that any valid C&C model has exactly one parent component. We have included this fact as a rule since it can be removed for advanced synthesis cases, e.g., when synthesizing a client and server architecture the server and all its clients are components on the top most level (see Section 5.5.2).

Translation Rule

Translation rule:

V1
```
fact oneRoot {
   one (Component - Component.subComponents)
}
```

Figure 5.19.: Translation rule V1 adding a fact about a single parent component.

The second rule V2 defined in Figure 5.20 creates a signature for each component found in any of the views. This signature has the multiplicity `lone` (0 or 1) since it is not known whether all components are contained in a satisfying C&C model. The translation of all six views shown in Figures 5.1, 5.2, and 5.3 produces the result shown in the lower part of Figure 5.20.

Translation Rule

Translation rule:

V2
```
// all components used in views
```
$\forall cmp \in \bigcup_{v \in views} v.Cmps$:
```
lone sig cmp.name extends Component {}
```

Result of application to views shown in Figures 5.1, 5.2, and 5.3:

Alloy

```
1 // all components used in views
2 lone sig RotationalJoint extends Component {}
3 lone sig Body extends Component {}
4 lone sig Actuator extends Component {}
5 lone sig Sensor extends Component {}
6 lone sig ServoValve extends Component {}
7 lone sig Cylinder extends Component {}
8 lone sig Joint extends Component {}
9 lone sig JointLimiter extends Component {}
```

Figure 5.20.: Translation rule V2 for components.

The third rule V3 defined in Figure 5.21 creates a singleton signature representing each port name occurring in any view. The result produced by translation rule V3 on the set of views depicted in Figures 5.1, 5.2, and 5.3 is shown in the lower part of Figure 5.21.

Translation Rule

Translation rule:

V3 `// all port names used in views`
$\forall pName \in \bigcup_{v \in views} v.PNames$:
`one sig` $pName$ `extends PortName {}`

Result of application to views shown in Figures 5.1, 5.2, and 5.3:

Alloy

```
1 //  all port names used in views
2 one sig val1 extends PortName {}
3 one sig f1 extends PortName {}
4 one sig val2 extends PortName {}
5 one sig f2 extends PortName {}
6 one sig angle extends PortName {}
```

Figure 5.21.: Translation rule V3 for port names.

The rule V4 defined in Figure 5.22 creates a singleton signature representing port types occurring in the views. The result produced by the translation rule V4 on the set of views depicted in Figures 5.1, 5.2, and 5.3 is shown in the lower part of Figure 5.22.

Translation Rule

Translation rule:

V4 `// all types used in views`
$\forall type \in \bigcup_{v \in views} v.Types$:
`one sig my_` $type$ `extends Type {}`

Result of application to views shown in Figures 5.1, 5.2, and 5.3:

Alloy

```
1 // all types used in views
2 one sig my_int extends Type {}
3 one sig my_float extends Type {}
```

Figure 5.22.: Translation rule V4 for port data types. We add the prefix `"my_"` to all type names to avoid name clashes, e.g., with Alloy's built-in signature `int`.

An overview of the translation rules that translate a single C&C view into an Alloy predicate expressing the view's semantics is shown in Figure 5.18. The resulting predicate consists of statements about the existence of all components shown in the view (rule P1), their possible independence (rule P2) and containment (rule P3), a list of statements about the ports of components, and statements about component connections (rule P4).

As an example for the translation of a C&C view to an Alloy predicate consider the view `BodySensorOut` shown in Figure 5.23.

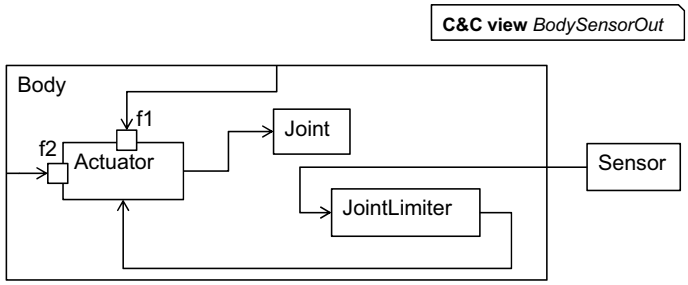


Figure 5.23.: The C&C view `BodySensorOut` as shown in Figure 5.2 (b).

Translation rule P1 shown in Figure 5.24 adds statements for each component in the view that it has to exist in an Alloy instance satisfying the view's predicate.

Translation Rule

Translation rule:

P1 `// all components shown in the view exist`

$\forall cmp \in view.Cmps$:

`one` $cmp.name$

Result of application to view `BodySensorOut` shown in Figure 5.23:

Alloy

```
1 // all components shown in the view exist
2 one Body
3 one Actuator
4 one Sensor
5 one JointLimiter
6 one Joint
```

Figure 5.24.: Translation rule P1 regarding the existence of components.

Translation rule P2 shown in Figure 5.25 instantiates predicates to ensure that components shown on the same level and contained by the same parent are enforced to be

independent, i.e., not contained in each other in any satisfying C&C model. The first part of the translation handles the top-level components which are not contained in any other component shown in the view. In the view `BodySensorOut` these are the components `Body` and `Sensor`. The second part handles the sets of immediate subcomponents of components shown in the view.

Translation Rule

Translation rule:

P2 `// top-level components are pairwise independent`

$$\textbf{let } topCmps = \{cmp \in view.Cmps \mid \nexists cmp' \in view.Cmps : cmp \in cmp'.subs\} \textbf{ in}$$

`independentSet[` $\{\!|cmp.name|\!\}^{+}_{cmp \in topCmps}$ `]`

`// sibling components are pairwise independent`

$$\forall cmp \in view.Cmps, iSet = \{sc \mid sc \in cmp.subs\} \neq \emptyset :$$

`independentSet[` $\{\!|sc.name|\!\}^{+}_{sc \in iSet}$ `]`

Result of application to view `BodySensorOut` shown in Figure 5.23:

Alloy

```
1 // top-level components are pairwise independent
2 independentSet[Body + Sensor]
3 // sibbling components are pairwise independent
4 independentSet[Actuator + Joint + JointLimiter]
```

Figure 5.25.: Translation rule P2 regarding independence of components. The operator $\{\!|body|\!\}^{sep}_{elems}$ executes the statement *body* with all elements from the set *elems* and separates the results with the separator *sep* (see Appendix B).

Translation rule P3 shown in Figure 5.26 instantiates the predicate `contains` for every component and its immediate subcomponents. If a component has no subcomponents nothing is generated by this rule. In the C&C view `BodySensorOut` only component `Body` has subcomponents. The corresponding instantiation of the predicate for this component is shown in the listing at the bottom of Figure 5.26.

Translation rule P4 shown in Figure 5.27 iterates over all components of the view and instantiates predicates based on the completeness of the port's type and name information. The example view `BodySensorOut` shows two incoming ports on component `Actuator` with names `f1` and `f2` but with unknown types.

Translation Rule

Translation rule:

P3 `// component containment relation`

$\forall parent \in \{parent \in view.Cmps \mid \exists child \in parent.subs\}:$

`contains[` $parent.name$ `,` $\{child.name\}^{+}_{child \in parent.subs}$ `]`

Result of application to view `BodySensorOut` shown in Figure 5.23:

Alloy

```
1 // containment relation
2 contains[Body, Actuator + Joint + JointLimiter]
```

Figure 5.26.: Translation rule P3 regarding containment of components.

Translation Rule

Translation rule:

P4 `// ports of components`

$\forall cmp \in view.Cmps:$

`// ports of component` $cmp.name$

$\forall port \in \{port \in cmp.ports \mid port.name = \bot\}:$

`unnamedPort[` $cmp.name$ `,` $port.dir$ `,` $port.type$ `]`

$\forall port \in \{port \in cmp.ports \mid port.type = \bot\}:$

`untypedPort[` $cmp.name$ `,` $port.dir$ `,` $port.name$ `]`

$\forall port \in \{port \in cmp.ports \mid port.type \neq \bot \wedge port.name \neq \bot\}:$

`portOfComponent[` $cmp.name$ `,` $port.dir$ `,` $port.type$ `,` $port.name$ `]`

Result of application to view `BodySensorOut` shown in Figure 5.23:

Alloy

```
1 // ports of components Actuator
2 untypedPort[Actuator, IN, f1]
3 untypedPort[Actuator, IN, f2]
```

Figure 5.27.: Translation rule P4 regarding ports of components.

Translation rule P5 shown in Figure 5.28 has four parts that handle abstract connectors. Each exclusive part instantiates a corresponding predicate about the connectedness with all information available from the abstract connector. The example C&C view `BodySensorOut` contains three abstract connectors with no information about the source or target ports given (ll. 2-4, listing in Figure 5.28) and two abstract connectors with a known target port but unknown source port (ll. 5-6).

Translation Rule

Translation rule:

P5 `// connections between components`

$$\forall c \in \{c \in view.AbsCons \mid c.srcPort = \bot \wedge c.tgtPort = \bot\} :$$
$$\texttt{connected}[c.srcCmp, c.tgtCmp]$$
$$\forall c \in \{c \in view.AbsCons \mid c.srcPort \neq \bot \wedge c.tgtPort = \bot\} :$$
$$\texttt{connectedWithReceiverPortName}[c.srcCmp, c.tgtCmp, c.tgtPort]$$
$$\forall c \in \{c \in view.AbsCons \mid c.srcPort = \bot \wedge c.tgtPort \neq \bot\} :$$
$$\texttt{connectedWithSenderPortName}[c.srcCmp, c.srcPort, c.tgtCmp]$$
$$\forall c \in \{view.AbsCons \mid c.srcPort \neq \bot \wedge c.tgtPort \neq \bot\} :$$
$$\texttt{connectedWithPortNames}[c.srcCmp, c.srcPort, c.tgtCmp, c.tgtPort]$$

Result of application to view `BodySensorOut` shown in Figure 5.23:

Alloy

```
1 // connections between components
2 connected[Actuator, Joint]
3 connected[Sensor, JointLimiter]
4 connected[JointLimiter, Actuator]
5 connectedWithReceiverPortName[Body, Actuator, f1]
6 connectedWithReceiverPortName[Body, Actuator, f2]
```

Figure 5.28.: Translation rule P5 regarding connections between components.

5.3.5. Synthesis Predicate and Alloy Command

Finally, we construct a predicate `specification` representing the specification's propositional formula over the views' predicates. This predicate is exactly the propositional formula of the views specification S. The translation of a specification S into the Alloy predicate `specification` is straight forward since the names of the views are the names of the Alloy predicates representing the semantics of the views (see Figure 5.18). The predicate `specification` for specification S_1 from the robotic arm example in

Section 5.1.1 is shown in Listing 5.29 (ll. 1-5). In our implementation we have developed a user friendly dialog to compile the specification S based on the views of the synthesis problem (see Section 5.6.1 and Appendix D).

Alloy

```
1 pred specification {
2    (RJFunction) and (RJStructure) and
3    (BodySensorIn or BodySensorOut) and
4    (SensorConnections) and (not ASDependence)
5 }
6
7 run specification for 6 but exactly 20 Port, 8 Component
```

Listing 5.29: The Alloy predicate `specification` representing the views specification S_1 introduced in Section 5.1.1 and the Alloy command to find a satisfying C&C model.

We run the module defined above with an Alloy run command to find an instance that satisfies the predicate `specification`. The Alloy run command from the robotic arm example is shown in Listing 5.29 (l. 7). Please note that Alloy analysis must be done within a user-defined, given scope, which specifies an upper bound for the number of instances per signature. The default scope `6` (l. 7) is an upper bound for the signatures `Type` and `PortName`. We automatically calculate the upper bound for the number of components. The upper bound for signature `Component` is the number of unique component names contained in all views occurring in the specification (in this example `8`). The upper bound for the total number of ports and the maximal number of ports per component, however, cannot be derived from the specification. We currently let the user choose the scope for ports manually. In the example shown in Listing 5.29 the chosen upper bound for the number of ports is `20`.

If an Alloy instance is found, we translate it back to the problem domain, that is, to a C&C model specified in MontiArc. However, if an Alloy instance is not found, in the general case, we do not know whether the specification could be satisfiable in a larger scope, that is, using more ports. As stated before, our solution is sound but incomplete (although it is complete within the given scopes).

The translation back is linear in the size of the solution and we describe it in Section 5.3.6.

5.3.6. Translation of Alloy Instances into C&C Models

An Alloy instance consists of the signatures, fields and relations specified by the Alloy module. Each signature is a set containing atoms within the scope of the signature. Each field and relation is translated into a set of tuples of atoms. All Alloy modules generated for C&C views synthesis include the signature definitions and facts of the C&C model metamodel described in Section 5.3.2. All generated instances thus have a common form described in Definition 5.30.

Definition 5.30 (Alloy instances of C&C views synthesis)**.** An Alloy instance computed for C&C views synthesis consists of the sets of atoms

- *Component* of atoms for components,
- *Port* of atoms for ports,
- $Direction = \{IN, OUT\}$ of atoms for directions of ports,
- *Type* of atoms for port types, and
- *PortName* of atoms for port names,

and the relations

- $subComponents \subseteq (Component \times Component)$ of direct subcomponents,
- $parent \subseteq (Component \times Component)$ the inverse of relation *subComponents*,
- $ports \subseteq (Component \times Port)$ of components and their ports
- $owner \subseteq (Port \times Component)$ the inverse of relation *ports*,
- $direction \subseteq (Port \times Direction)$ of ports and their directions,
- $type \subseteq (Port \times Type)$, of ports and their types,
- $name \subseteq (Port \times PortName)$ of ports and their names, and
- $receivingPorts \subseteq (Port \times Port)$ of ports and the ports they are connected to.
- $sendingPort \subseteq (Port \times Port)$ the inverse of relation *receivingPorts*.

△

Both the Alloy metamodel described in Section 5.3.2 and the Alloy instances for C&C views synthesis contain redundant relations, e.g., the relation *subComponents* and its inverse relation *parent*. These redundancies help to make the definitions inside Alloy and in the translation rules more intuitive. From our experience and experiments with the formulation of the synthesis problem inside Alloy it is not clear whether the redundancies improve or deteriorate the performance of C&C views synthesis (see Section 5.7.2).

The Alloy instance corresponding to the C&C model from Figure 5.4 for the C&C views specification S_1 from Section 5.1.1 is shown in the concrete textual syntax of Alloy instances in Listing 5.31. The sets of atoms are listed in the order as specified in Definition 5.30 in lines 1-7 of the listing. The computed relations of the Alloy instance are shown in lines 9-36. We have omitted the two relations $parent \subseteq (Component \times Component)$ and $sendingPort \subseteq (Port \times Port)$ since they are not required for the translation back into a MontiArc C&C model. These relations are the inverse relations of the relations $subComponents \subseteq (Component \times Component)$ and $receivingPorts \subseteq (Port \times Port)$, respectively. The instance shown in Listing 5.31 contains both the relation $ports \subseteq (Component \times Port)$ and its inverse the relation $owner \subseteq (Port \times Component)$ since we use the relation *owner* in the translation rules for the translation of connectors from a computed Alloy instance into a MontiArc C&C model.

Alloy Instance

```
Component = {Actuator, Body, Cylinder, Joint, JointLimiter,
             RotationalJoint, Sensor, ServoValve}
Port = {P0, P1, P10, P11, P12, P13, P14, P15, P16, P17, P18, P19,
        P2, P3, P4, P5, P6, P7, P8, P9}
Direction = {IN, OUT}
Type = {my_float, my_int}
PortName = {PortName0, angle, f1, f2, val1, val2}

subComponents = {Body->Actuator, Body->Joint, Body->JointLimiter,
  RotationalJoint->Body, RotationalJoint->Cylinder,
  RotationalJoint->Sensor, RotationalJoint->ServoValve}
ports = {Actuator->P10, Actuator->P11, Actuator->P12,
  Actuator->P8, Actuator->P9, Body->P16, Body->P5, Body->P6,
  Body->P7, Cylinder->P17, Cylinder->P18, Cylinder->P19,
  Joint->P14, JointLimiter->P13, JointLimiter->P15,
  RotationalJoint->P0, Sensor->P2, Sensor->P3,
  ServoValve->P1, ServoValve->P4}
owner = {P0->RotationalJoint, P1->ServoValve, P10->Actuator,
  P11->Actuator, P12->Actuator, P13->JointLimiter, P14->Joint,
  P15->JointLimiter, P16->Body, P17->Cylinder, P18->Cylinder,
  P19->Cylinder, P2->Sensor, P3->Sensor, P4->ServoValve, P5->Body,
  P6->Body, P7->Body, P8->Actuator, P9->Actuator}
direction = {P0->IN, P1->IN, P10->IN, P11->IN, P12->OUT, P13->IN,
  P14->IN, P15->OUT, P16->OUT, P17->IN, P18->IN, P19->OUT, P2->OUT,
  P3->OUT, P4->OUT, P5->IN, P6->IN, P7->IN, P8->IN, P9->IN}
type = {P0->my_int, P1->my_int, P10->my_float, P11->my_float,
  P12->my_int, P13->my_float, P14->my_int, P15->my_float,
  P16->my_float, P17->my_float, P18->my_int, P19->my_int,
  P2->my_float, P3->my_int, P4->my_int, P5->my_int, P6->my_int,
  P7->my_float, P8->my_int, P9->my_int}
name = {P0->angle, P1->f2, P10->f1, P11->angle, P12->PortName0,
  P13->val1, P14->angle, P15->val2, P16->val1, P17->angle,
  P18->val1, P19->val2, P2->val1, P3->val2, P4->f1, P5->f2,
  P6->PortName0, P7->angle, P8->f2, P9->val2}
receivingPorts = {P0->P1, P12->P14, P15->P11, P16->P17, P19->P5,
  P2->P7, P3->P18, P4->P6, P5->P8, P6->P9, P7->P10, P7->P13}
```

Listing 5.31: An instance generated by the Alloy Analyzer for the Alloy module generated from the C&C views specification S_1. The computed instance corresponds to the C&C model shown in Figure 5.4. The relations $parent \subseteq (Component \times Component)$ and $sendingPort \subseteq (Port \times Port)$ are not shown in the listing since they are not used in the translation of Alloy instances into MontiArc models.

The translation of Alloy instances into MontiArc C&C models starts with the translation rule I1 shown in Figure 5.32. The translation rule determines the single atom $root \in Component$ for which the relation *subComponents* does not define a parent. It then executes the translation rule I2 for components with the parameter *root*. We use the join operator $\bowtie$ as an equivalent to the dot-join operator of Alloy [Jac06, Section 3.4.3] (see also Appendix A). The atom $parent \in Component$ is treated by the operator as the unary relation containing the single element *parent*. The join of the unary relation $\{parent\}$ with the binary relation *subComponents* results in the unary relation $\{cmp \in Component \mid (parent, cmp) \in subComponents\}$, i.e., the set of all subcomponents of the component *parent*.

Translation Rule

Translation rule:

I1 **let** $root = THE\ cmp \in Component :$
$\nexists parent \in Component : cmp \in parent \bowtie subComponents$ **in**
executeRule(I2 *root***)** *see Figure 5.33 for body of rule I2*

Figure 5.32.: Translation rule I1 for the translation of an Alloy instance into a C&C model in the concrete syntax of MontiArc. This rule selects the parent component of the computed Alloy instance and translates it into concrete MontiArc syntax by executing rule I2.

The translation rule I2 is shown in the upper part of Figure 5.33. This translation rule is parametrized with an atom $cmp \in Component$. The rule creates a MontiArc component definition for the component represented by the atom (see Section 2.2 on how to model C&C models using MontiArc). All Alloy atoms are basic entities with no further properties. Atoms are identified only by their name. Thus, using the element $cmp \in Component$ in a translation rule prints the name of the atom as shown in the result of the execution of rule I2 with the parameter $RotationalJoint \in Component$ in the lower part of Figure 5.33. The translation rule I2 executes rule I3 in case the translated component *cmp* has ports. It executed itself recursively for all subcomponents of the component *cmp* and finally executes rule I4 to print all connectors defined inside the component *cmp*.

Translation Rule

Translation rule with parameter $cmp \in Component$:

I2 `component` *cmp* `{`

```
    if (∃p ∈ cmp ⋈ ports) then
        executeRule(I3 cmp)   see Figure 5.34 for body of rule I3
    // subcomponents of component cmp
∀sub ∈ cmp ⋈ subComponents :
    executeRule(I2 sub)   recursive execution of this rule
    // connectors of component cmp
    executeRule(I4 cmp)   see Figure 5.35 for body of rule I4
}
```

Result of application to the Alloy top component of the instance shown in Listing 5.31 (parameter $cmp = RotationalJoint \in Component$):

MontiArc

```
1  component RotationalJoint {
2    see result of execution of rule I3
3
4    // subcomponents of component RotationalJoint
5    execution of rule I2 (recursive application of rule)
6    for components Body, Cylinder, Sensor, and ServoValve
7
8    // connectors of component RotationalJoint
9    see result of execution of rule I4
10 }
```

Figure 5.33.: Translation rule I2 for the translation of a component $cmp \in Component$ of an Alloy instance into a component in the concrete syntax of MontiArc.

The translation rule I3 is parametrized with an atom $cmp \in Component$ and creates MontiArc declarations for the input and output ports of the component from the C&C model corresponding to *cmp*. The rule is shown in the upper part of Figure 5.34. The direction of ports is defined by the preceding keyword `in` or `out`. For a port $p \in Port$ the joins $p \bowtie type$ and $p \bowtie name$ each result in a single element. In the translation rule I2 we were able to use the atoms $cmp \in Component$ directly to print the names of the components they represent. This is different for the atoms $p \in Port$, since these have names on their own, e.g., `P0` or `P12`, as can be seen in the Alloy instance in Listing 5.31. The names of ports $p \in Port$ of the Alloy instance in their MontiArc representation are given as the second elements of the relation $name \subseteq (Port \times PortName)$. Thus, the statement $p \bowtie name$ prints the port name in the translation rule I3 as demonstrated in the listing in the lower part of Figure 5.34.

As part of rule I3 the prefix `my_` (added to type names in the translation rule V4 defined in Figure 5.22) is removed from the port types. We denote this simple String manipulation informally as the function *remPrefix*.

Translation Rule

Translation rule with parameter $cmp \in Component$:

I3
```
// ports of component cmp
port
{ if(p ⋈ direction = IN)
    in remPrefix(p ⋈ type) p ⋈ name
  else
    out remPrefix(p ⋈ type) p ⋈ name}_{p∈cmp⋈ports}' ;
```

Result of application to the Alloy instance shown in Listing 5.31 with the parameter $cmp = RotationalJoint \in Component$:

MontiArc
```
1 // ports of component RotationalJoint
2 port
3   in float val2;
```

Figure 5.34.: Translation rule I3 for the translation of the ports of a component $cmp \in Component$ of an Alloy instance to the concrete syntax of MontiArc. The function *remPrefix* removes the prefix `my_` added to type names in the translation rule V4 from Figure 5.22.

The translation rule I4 shown in Figure 5.35 creates the declarations of MontiArc connectors in the C&C model represented by the computed Alloy instance. The rule is again parametrized with an atom $cmp \in Component$. The first two **let** ... **in** statements introduce the local variables *cmpPorts* of ports of the component *cmp* and *subCmpPorts* of all ports of the direct subcomponents of the component *cmp*. The rule consists of two outer quantifications, one over each of the introduced sets of ports. The two nested quantifications range first over the ports of the component *cmp* and then over the ports of its subcomponents, that are connected to the port selected in the parent quantification. A join of the outer quantified port with the relation *receivingPorts* yields only the connected ports. The intersection with the ports of the component or the ports of the subcomponents leaves only the ports relevant for one of the four cases of connectors. There are four cases for MontiArc `connect` statements since ports of the parent component *cmp* are not qualified with their component name, while the ports of subcomponents are.

Translation Rule

Translation rule with parameter $cmp \in Component$:

I4 **let** $cmpPorts = cmp \bowtie ports$ **in**
let $subCmpsPorts = cmp \bowtie subs \bowtie ports$ **in**
$\forall p_{cmp} \in cmpPorts :$
$\forall p \in cmpPorts \cap (p_{cmp} \bowtie receivingPorts) :$
connect $p_{cmp} \bowtie name$ -> $p \bowtie name$;
$\forall p \in subCmpsPorts \cap (p_{cmp} \bowtie receivingPorts) :$
connect $p_{cmp} \bowtie name$ -> $p \bowtie owner$. $p \bowtie name$;
$\forall p_{sub} \in subCmpPorts :$
$\forall p \in cmpPorts \cap (p_{sub} \bowtie receivingPorts) :$
connect $p_{sub} \bowtie owner$. $p_{sub} \bowtie name$ -> $p \bowtie name$;
$\forall p \in subCmpsPorts \cap (p_{sub} \bowtie receivingPorts) :$
connect $p_{sub} \bowtie owner$. $p_{sub} \bowtie name$ -> $p \bowtie owner$. $p \bowtie name$;

Result of application to the Alloy instance shown in Listing 5.31 with the parameter $cmp = RotationalJoing \in Component$:

MontiArc

```
1  connect val2 -> ServoValve.val2;
2  connect Body.angle -> Cylinder.angle;
3  connect Cylinder.f1 -> Body.PortName0;
4  connect Sensor.val1 -> Body.val2;
5  connect Sensor.val2 -> Cylinder.f2;
6  connect ServoValve.PortName0 -> Body.val1;
```

Figure 5.35.: Translation rule I4 for the translation of the connectors of a composed component $cmp \in Component$ of an Alloy instance to the concrete syntax of MontiArc.

5.4. Advanced Features

We have extended our work on C&C model synthesis with several advanced features to support engineers when creating views and views specifications. An extension of the views language with stereotypes (see Section 5.4.1) allows users to add additional knowledge to views beyond the C&C views language introduced in Chapter 3.6. Another extension is adding support for library components to C&C views synthesis (see Section 5.4.2). We have identified several specification patterns in views specifications and introduce them in Section 5.4.3.

5.4.1. Handling Language Variability in C&C Views

The C&C views language as defined in Definition 3.6 has a built-in mechanism for language variability based on stereotypes. We demonstrated the variability with two language extensions on the level of the abstract syntax (see Section 3.3.1), the concrete syntax (see Section 3.6.5), and the semantics in terms of the views satisfaction relation (see Section 3.4.1). In this section we show how we support these additional language features in our translation of C&C views into Alloy.

We have extended our translation of C&C views into Alloy with support for marking components as interface-complete and as atomic. The engineer marks components in a view using stereotypes. The additional constraint is interpreted at the level of the view, e.g., a component that is marked as atomic is only required to be atomic in a C&C model satisfying the view.

Interface-complete components An engineer can strengthen a view by declaring some of the components mentioned in it as interface-complete (technically, using the stereotype «`interfaceComplete`» in front of a component). This is useful when the engineer knows the complete interface of a component, e.g., some subcomponents or their hierarchy are either not relevant in the view or not completely known.

For example, consider adding the stereotype «`interfaceComplete`» to the component `Sensor` in the view `SensorConnections` shown in Figure 5.36. In a synthesized C&C model satisfying the view `SensorConnections`, the component `Sensor` must have exactly the set of ports (identified by their name) defined in this view.

The additional Alloy predicate we use to support interface-complete components is shown in Listing 5.37. We instantiate this predicate with every component that is specified as interface-complete in a view.

To support the stereotype «`interfaceComplete`», we have adapted our basic translation of C&C views into Alloy from Section 5.3.4 by adding the new rule P4a to the set of translation rules. The new rule is executed inside the body of the predicate describing a view (see Figure 5.18). Rule P4a is shown in Figure 5.38 with an example application to the view `SensorConnections` from Figure 5.36 with the interface-complete component `Sensor`.

The stereotype «`interfaceComplete`» is applicable to any component and specifies nothing about the component's decomposition.

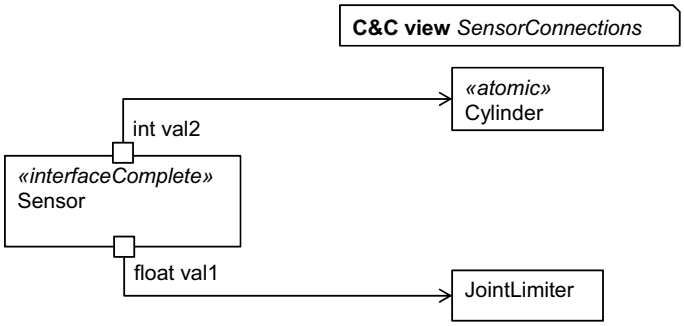


Figure 5.36.: The view `SensorConnections` adapted from Figure 5.3 with component `Sensor` marked as interface-complete and component `Cylinder` marked as atomic.

Alloy

```
1 pred interfaceComplete[cmp: Component, portNames: set PortName] {
2   cmp.ports.name = portNames
3 }
```

Listing 5.37: Predicate to specify that a component is interface-complete, technically, by stating that the set of its port names, as appearing in the view, is exactly its complete set of port names.

Translation Rule

Translation rule:

P4a $\forall cmp \in \{cmp \in view.Cmps \mid \text{«}\texttt{interfaceComplete}\text{»} \in cmp.stereotypes\}$:
`// component` *cmp.name* `is interface-complete`
`if(`*cmp.ports* $\neq \emptyset$**`) then`**
`interfaceComplete[`*cmp.name*`,` $\{\!|p.name|\!\}^{+}_{p \in cmp.ports}$`]`
`else`
`interfaceComplete[`*cmp.name*`, none]`

Result of application to view `SensorConnections` shown in Figure 5.36:

Alloy

```
1 // component Sensor is interface-complete
2 interfaceComplete[Sensor, val1 + val2]
```

Figure 5.38.: Translation rule P4a handling components marked as interface-complete.

Atomic components Based on existing knowledge or requirements an engineer may mark a component in a view as atomic (technically, using the stereotype «atomic» in front of a component). Atomic components are not further decomposed in C&C models.

The additional Alloy predicate we use to support atomic components is shown in Listing 5.39. The predicate `atomicComponent` parametrized with component `cmp` states that the subcomponent relation of component `cmp` is empty (l. 3) and that the component has no connection starting from its incoming ports, i.e., no internal connectors (l. 5). We instantiate this predicate with every component that is specified as atomic in a view.

Alloy

```
1 pred atomicComponent[cmp: Component] {
2   // no children
3   no cmp.subComponents
4   // no internal connectors (starting from incoming ports)
5   no (cmp.ports & direction.IN).receivingPorts
6 }
```

Listing 5.39: Predicate to specify that a component is atomic, technically, by stating that the component has no subcomponents in a satisfying C&C model and no internal connectors.

We have adapted our basic translation of C&C views into Alloy from Section 5.3.4 by adding the new rule P3a to the set of translation rules executed inside the body of the predicate describing a view (see Figure 5.18). Rule P3a is shown in Figure 5.40 with an example application to the view `SensorConnections` from Figure 5.36 with the atomic component `Cylinder`.

Translation Rule

Translation rule:

P3a $\forall cmp \in \{cmp \in view.Cmps \mid \text{«atomic»} \in cmp.stereotypes\}$:
`// component` *cmp.name* `is atomic`
`atomicComponent[`*cmp.name*`]`

Result of application to view `SensorConnections` from Figure 5.36:

Alloy

```
1 // component Cylinder is atomic
2 atomicComponent[Cylinder]
```

Figure 5.40.: Translation rule P3a about components marked as atomic.

5.4.2. Library Components

Most software systems reuse library components, pre-defined or existing components adopted from other systems. Thus, it is crucial that our technique would allow the architect to import such components and apply an integrated synthesis solution.

Specifically, our C&C model synthesis supports the integration of library components or similar components at two levels. First, the architect can extend the specification with a list of imported library component definitions. A component definition is complete: it specifies the complete interface (port names and types) of the component. If a component definition for the component `Cmp` is imported, we add its interface (and the requirement that it is complete) as an additional constraint to the generated Alloy module. This ensures that a synthesized C&C model that uses `Cmp` would be consistent with its interface and use it as is.

As an example, consider the `ServoValve` component in the view `RJStructure` shown in Figure 5.1 (b) to be an imported library component. In the library, the complete interface of `ServoValve` is given. Importing it to the specification S_1 ensures its interface is used as is in the synthesized C&C model and rules out solutions that put other components that are mentioned in the views as its subcomponents. For example, a solution where `ServoValve` contains `Sensor` will not be possible.

To support library components we add a library component specification *lib* as defined in Definition 5.41 to the input for the translation on the same level as the set *views*. Each component $cmp \in lib.Cmps$ contains the definition of a library component, e.g., the set *cmp.ports* contains the complete set of ports of the component *cmp*.

Definition 5.41 (Library components specification)**.** A library components specification is a C&C model *lib* as defined in Definition 2.2 (not necessarily with a single top component). △

In addition to adding the library components specification to the input, we also add rule V5 shown in Figure 5.42 to the translation. This translation rule reuses the predicates `atomicComponent` (see Listing 5.39) and `interfaceComplete` (see Listing 5.37). Library components are not part of a single C&C view, but of a C&C views synthesis problem. Multiple views may refer to the same library component. The translation rule V5 is added to the set of rules executed for the generation of the C&C views synthesis Alloy module on the same level as rules V1 to V4 (see Figure 5.18).

The translation rule for library components only adds constraints to the fact `libraryComponents` if the library component is mentioned in at least one C&C view. Otherwise the library component is not relevant for the synthesis problem. But although a library component is mentioned in a view and added to the set of library components, it is still possible that the component is not part of an instance satisfying the C&C views specification.

Instantiating the Alloy predicate `interfaceComplete` with any component as parameter `cmp` forces the existence of that component in an Alloy instance since the predicate forces the expression `cmp.ports.name` to evaluate to the parameter `portNames` (see Listing 5.37) and it thus forces the component `cmp` to exist. To prevent this side

Translation Rule

Translation rule:

V5 $\forall type \in lib.Types \setminus \bigcup_{v \in views} v.Types$:
`one sig my_` $type$ `extends Type {}`
$\forall pName \in lib.PNames \setminus \bigcup_{v \in views} v.PNames$:
`one sig` $pName$ `extends PortName {}`

`fact libraryComponents {`
$\forall cmp \in lib.Cmps \cap \bigcup_{v \in views} v.Cmps$:
`atomicComponent[`$cmp.name$`]`
if($cmp.ports \neq \emptyset$**) then**
`some` $cmp.name$ `implies (`
`interfaceComplete[`$cmp.name$`,` $\{p.name\}^{+}_{p \in cmp.ports}$`] and`
`{portOfComponent[`$cmp.name$`,` $port.direction$`,` $port.type$`,`
$port.name$`]` $\}^{\text{and}}_{p \in cmp.ports}$
`)`
else
`some` $cmp.name$ `implies interfaceComplete[`$cmp.name$`, none]`
`}`

Result of application to the C&C model `ServoValve` containing the library component `ServoValve` shown in Figure 5.43:

Alloy

```
1  one sig forceLimit extends PortName {}
2  one sig torque extends PortName {}
3
4  fact libraryComponents {
5    atomicComponent[ServoValve]
6    some ServoValve implies (
7      interfaceComplete[ServoValve, forceLimit + torque] and
8      portOfComponent[ServoValve, IN, my_float, forceLimit] and
9      portOfComponent[ServoValve, OUT, my_float, torque]
10   )
11 }
```

Figure 5.42.: Translation rule V5 to create a fact about library components. The rule also creates signatures for port names and types required for the definition of the library components if not previously defined.

effect of the predicate `interfaceComplete`, the translation rule V5 introduces an implication depending on the existence of an instance of each library component. The instantiation of the predicate `atomicComponent` does not have a similar side effect.

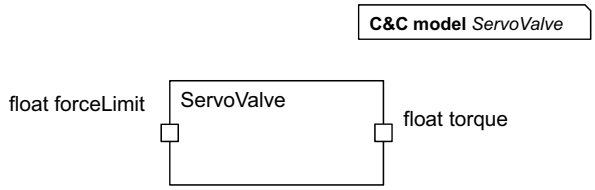


Figure 5.43.: The library component `ServoValve` with the input port `forceLimit` and the output port `torque` both of type `float`.

The result of an application of the rule V5 to the C&C views *views* from the views specification $S1$ and the C&C model *lib* containing the library component `ServoValve` from Figure 5.43 is shown in the lower part of Figure 5.42.

Please note that the synthesis considers an imported component as a black-box: it uses its interface and needs no knowledge of its subcomponents. This separation is meant to support encapsulation and modularity: as long as the interface is kept fixed, the implementation of imported library components can be replaced without affecting the synthesized C&C model.

Finally, please note that both features, importing library components and declaring components in views as interface-complete, not only extend the usefulness and expressive power of C&C views, but also further constrain the specification. As part of our evaluation (see Section 5.6) we examine the effect of the use of library and interface-complete components on the performance of synthesis for various example specifications.

5.4.3. Specification Patterns

The use of propositional formulas in C&C views specifications makes them very expressive. However, for the architect who constructs the specification, using only low level basic propositional connectives may be inconvenient and error prone. Thus, we look for higher-level specification patterns, which can be used to express the required semantics more intuitively, and can be reused across different specifications.

Based on our experience with creating specifications, some examples of simple patterns are: [ALT], given a set of views, specifying that the synthesized C&C model must satisfy at least one of the alternative views in the set; [ONEALT], given a set of views, specifying that the synthesized C&C model must satisfy exactly one of the alternative views in the set; [IMP], given two views, if the C&C model satisfies the first, it must also satisfy the second; and [NOCOMP], a given component should not be present in the synthesized C&C model. Please note that the last three patterns depend on the use of negation in the specification language.

As an example, recall specification S_1 (Section 5.1.1). Assume that the architect responsible for `Sensor` knows that if it is located outside `Body`, then it must use an inner amplifier `Amplifier`. To express this knowledge, the architect can create the view `SensorHasAmp` shown in Figure 5.44 (a) and add the implication `IMP(BodySensorOut, SensorHasAmp)` to the specification's propositional formula (as another conjunct). To make sure the amplifier is not used when it is not necessary, the architect can create the view `Amp` shown in Figure 5.44 (b), and add the implication `IMP(BodySensorIn, ¬Amp)` (since the view `Amp` consists of the single component `Amplifier`, the expression `¬Amp` used here is an instance of the pattern [NOCOMP] mentioned above).

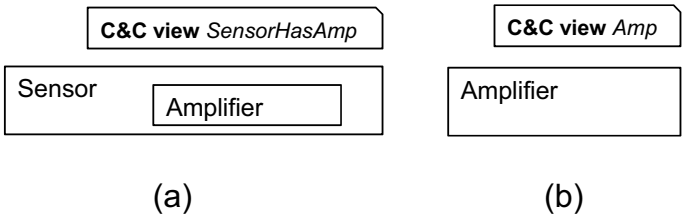


Figure 5.44.: The additional views `SensorHasAmp` and `Amp`, to be added to the specification S_1 with the implications `IMP(BodySensorOut,SensorHasAmp)` and `IMP(BodySensorIn,¬Amp)`.

As another example, the choice between `BodySensorIn` and `BodySensorOut` in S_1 can be strengthened by the architect to `ONEALT(BodySensorIn, BodySensorOut)` without loosing or adding possible implementations, that is, as a form of refactoring. The semantics of the two views entails that no C&C model can satisfy both and their use in the specification S_1 entails that at least one of them must be satisfied.

Specification patterns do not add expressive power. Rather, they are only meant to improve the readability and usability of C&C views specifications. Indeed, as part of our prototype implementation, we have developed friendly interfaces to add and edit the specification's propositional formula, including support for several patterns as shown in Figure 5.45. Our ideas on specification patterns for C&C model structures are inspired by previous works on patterns of temporal specifications (e.g., [DAC99]).

5.5. Synthesis with Architectural Styles

Architectural styles systematize successful architectural design practices in terms of constraints on architectural elements and their composition into systems [SG96]. Clements et al. [CBB+10] define an architectural style as "a specialization of element and relation types, together with a set of constraints on how they can be used". Some examples of well-known architectural styles include the pipe-and-filter style, the client-server style, and the layered style (an overview can be found in [TMD09, Chapter 4]).

Besides additional constraints, architectural styles often add a new vocabulary for elements in C&C models, e.g., *client*, *server*, or *layer*. Benefits of styles are the application

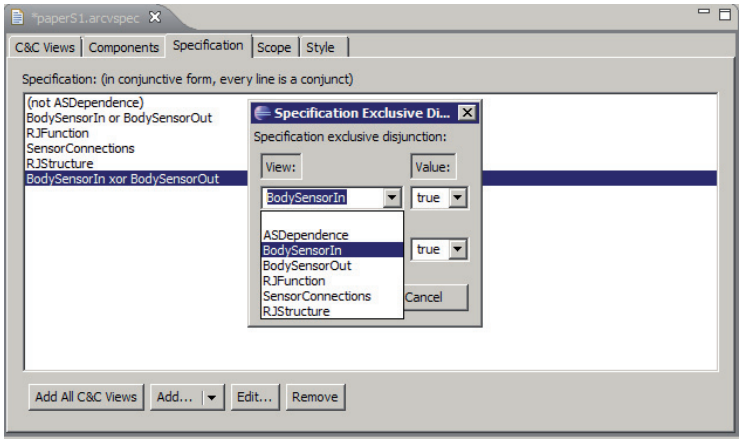


Figure 5.45.: The specification editor developed as part of our C&C views synthesis Eclipse plug-in. The screen capture shows the dialog to create a conjunct following the [ONEALT] pattern (`xor` in the screen capture).

of well-understood solutions to new problems and improved understanding conveyed by the problem specific terminology [TMD09]. For example, the architectural design environment AcmeStudio [wwwy] supports style-specific visualizations of C&C models.

We have extended C&C views synthesis with support for three architectural styles. As applicable, style specific constraints are added to the synthesis input so that the synthesized C&C model, if any is found, obeys the rules of the style. We give three examples below and show how the translation rules are adapted to support the styles. All three architectural styles are supported by our implementation.

5.5.1. Hierarchical Style

A C&C model in the *hierarchical style* has no directed cycles of connected components. The hierarchical style is important, because hierarchical architectures are suitable for behavioral synthesis: while the problem of synthesizing a finite-state distributed reactive system over a given architecture is in general undecidable, it is decidable for the class of hierarchical architectures [PR90].

To enforce the hierarchical style of the synthesized C&C model we add to the generated Alloy module a fact that requires that no directed cycles exist (see Listing 5.46). The implementation of the hierarchical style shown in Listing 5.46 does not require any changes to the translation rules shown in the translation overview in Figure 5.18. The new fact, signature, and predicate depicted in Listing 5.46 are added to the generic part of the translation.

The type of connections relevant for the hierarchical style are end-to-end connections between components. End-to-end connections between components are defined in the

Alloy

```
1  fact hierarchicalArchitecture {
2    no c: Component |
3      c in ^(Talker.talksTo).c
4  }
5
6  one sig Talker {
7    talksTo: Component -> Component
8  } {
9    all c1, c2: Component |
10     c2 in talksTo.c1 iff endToEndConnection[c1,c2]
11 }
12
13 pred endToEndConnection[senderC: one Component,
14                                    receiverC: one Component] {
15   some senderP: senderC.ports |
16     some receiverP: receiverC.ports |
17       no senderP.sendingPort and // no forward
18       no receiverP.receivingPorts and // no forward
19       receiverP in senderP.^receivingPorts
20 }
```

Listing 5.46: A fact for the hierarchical architecture style specifying that no component in the C&C model has a directed end-to-end feedback loop.

predicate `endToEndConnection` in Listing 5.46 in lines 13-20. The receiving port is in the transitive closure of the receiving ports of the sending port (l. 19), i.e., the ports are connected by a chain of directed connectors. The sending port is an endpoint if it has no incoming connector (l. 17) and the receiving port is an endpoint if it has no outgoing connector (l. 18). Examples for components satisfying the predicate `endToEndConnection` are the component pairs `Actuator` and `JointLimiter` or `RotationalJoint` and `Actuator` from the C&C model shown in Figure 5.47. The component pair `RotationalJoint` and `Body` does not satisfy the predicate since the incoming port of the component `Body` is not an endpoint.

The relation `talksTo` of the signature `Talker` is essentially a translation of the predicate `endToEndConnection` into a relation (see the equivalence defined in Listing 5.46, l. 10). This translation is needed since we are interested in the transitive closure of the end-to-end communication relation. Alloy has an operator for the transitive closure of relations but no similar concept for predicates. We thus translate the predicate to a relation to compute the transitive closure of the relation in line 3. We finally require that no component is in a (transitive) end-to-end communication relation with itself (l. 3) to enforce the hierarchical style.

The example of a C&C model shown in Figure 5.47 demonstrates that it is important to only consider end-to-end communication since the C&C model has a direct feedback communication cycle for component `Body`. The cycle is however resolved by the

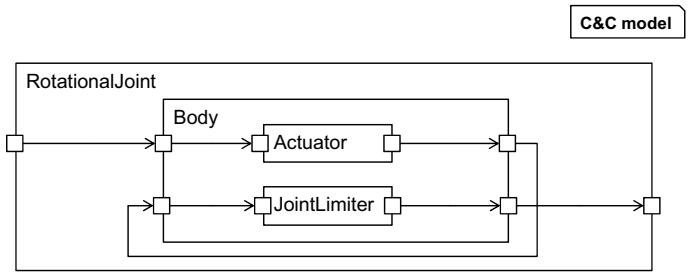


Figure 5.47.: An example of a C&C model in the hierarchical architecture style. Please note that the direct feedback connector of component `Body` is resolved inside the component and does not lead to a directed communication cycle.

decomposition of the component `Body` and thus the C&C model respects indeed the hierarchical style.

5.5.2. Client-Server Style

Our implementation of C&C views specification synthesis supports the *client-server style*, whose essence is to identify one of the components as a single server and to forbid any direct communication between clients. Also, the server and the clients are all assumed to be independent in terms of containment; a client is not contained within the server or another client etc. To integrate this style into our synthesis solution we add the identities of the server and the client components, as defined by the architect, to the specification. We add a client-server specification as defined in Definition 5.48.

Definition 5.48 (Client-server specification)**.** A client-server specification for a C&C views specification over the set of views *views* consists of *server* and *clients* where

- $server \in \{cmp.name \mid cmp \in \bigcup_{v \in views} v.Cmps\}$ is the name of the server component and
- $clients \subseteq \{cmp.name \mid cmp \in \bigcup_{v \in views} v.Cmps\} \setminus \{server\}$ is the set of names of the client components.

△

To make the knowledge of the server and the clients available in the Alloy module, we add the translation rule V6 shown in Figure 5.49 to the list of executed translation rules. The rule is executed at the same level as the rules V1 to V4 shown in the overview of the translation in Figure 5.18. The translation parameter *server* contains the name of the server component. Thus, the Alloy function `myServer` returns the Alloy atom that represents the server component. Similarly, the generated function `myClients` returns the atoms that represent the client components. An example of the application of rule V6 is shown the lower part of Figure 5.49.

Translation Rule

Translation rule:

V6

```
fun myServer : one Component {
   server
}

fun myClients : set Component {
   {client}^+_{client∈clients}
}
```

Result of application to a specification from the avionics system (see Section 5.6.2 and [wwwu] for the complete specification) with *server* = `Pilot_DM` and *clients* = {`PCM`, `Pilot_Display`}:

Alloy

```
1 fun myServer : one Component {
2   Pilot_DM
3 }
4
5 fun myClients : set Component {
6   PCM + Pilot_Display
7 }
```

Figure 5.49.: Translation rule V6 to create functions that return the server and the set of clients in the client-server style.

Translation Rule

Translation rule:

$V1_{cs}$

```
fact onlyServerAndClientsAreRoot {
   Component.subComponents =
                        Component - myServer - myClients
}
```

Figure 5.50.: Modification of translation rule V1 to $V1_{cs}$ for the client-server style. The fact states that all components except the client and the server components are subcomponents.

In addition, in the translation of C&C views into Alloy we replace the constraint of a single top component (translation rule V1 defined in Figure 5.19), with a constraint that specifies that the server and the clients are the only top components. This constraint is implemented in the translation rule $V1_{cs}$. By making the server and the clients the top components in the C&C model we also enforce that they are not contained in one another but may contain further subcomponents.

Translation Rule

Translation rule:

$V2_{cs}$

`// all components used in views`

$\forall cmp \in \bigcup_{v \in views} v.Cmps :$

if ($cmp.name \in \{server\} \cup clients$ **) then**

`one sig` $cmp.name$ `extends Component {}`

else

`lone sig` $cmp.name$ `extends Component {}`

Result of application to a specification from the avionics system (see Section 5.6.2 and [wwwu] for the complete specification) with *server* = `Pilot_DM` and *clients* = {`PCM`, `Pilot_Display`}:

Alloy

```
1 // all components used in views
2 lone sig FM extends Component {}
3 one sig PCM extends Component {}
4 one sig Pilot_Display extends Component {}
5 one sig Pilot_DM extends Component {}
```

Figure 5.51.: Translation rule $V2_{cs}$ for components. Components identified as client or server are required to exist in any synthesized C&C model.

The components identified as the server and the clients are required to exist in any synthesized C&C model. We thus modify the translation rule V2 defined in Figure 5.20 to the new translation rule $V2_{cs}$ shown in Figure 5.51. The translation rule V2 declares the Alloy signatures for C&C model components with the multiplicity `lone` for one or no instance. In case the component is identified as the server or the clients the new rule $V2_{cs}$ uses the multiplicity `one` for exactly one instance.

An application of the rule $V2_{cs}$ is shown in the lower part of Figure 5.51. The set of components from the views are the components `FM`, `Pilot_DM`, `PCM`, and `Pilot_Display`. In this example the component `Pilot_DM` is identified as the server and the components `PCM` and `Pilot_Display` are identified as clients. The components are from the avionics system example presented in Section 5.6.2. The complete set of views, a specification with client-server style, and a synthesized C&C model for

this example are available from [wwwu].

Finally, to enforce the communication constraints of the client-server style, we add the fact `clientServerCommunication` shown in Listing 5.52 to the generic part of the translation. The fact uses the parametrized predicate `immediatelyConnected-NoOrder` that defines whether two components in a C&C model are directly connected (in any direction). The fact enforces that all clients are connected to the server and that no two clients are connected with each other directly.

Alloy

```
1  fact clientServerCommunication {
2    all client : myClients |
3       immediatelyConnectedNoOrder[myServer, client] and
4       no c : (myClients-client) |
5        immediatelyConnectedNoOrder[client, c]
6  }
7
8  pred immediatelyConnectedNoOrder[c1: Component, c2: Component] {
9    some p : c1.ports |
10     ( p in c2.ports.receivingPorts or
11       p in c2.ports.sendingPort )
12 }
```

Listing 5.52: Excerpt from the Alloy code for the client-server style. The functions `myServer` and `myClients` are generated Alloy functions returning the server component and the client components respectively (see translation rule V6 shown in Figure 5.49).

Again, if a C&C model that satisfies the specification together with the additional restrictions exists, it is found. If not, it means that the semantics of the specification cannot be satisfied within the client-server style, e.g., without direct client to client connectors (or that it cannot be satisfied at all).

5.5.3. Layered Style

The *layered style* forces a partition of the system's components into a sequence of layers, and allows direct connectors on the top level only between consecutive layers.

Figure 5.53 shows a C&C model of the Avionics system (see description in Section 5.6.2) synthesized in the layered architectural style. All components that correspond to layers are marked with the stereotype «`layer`». The complete views, the C&C views specification, and a synthesized C&C model conforming to the layered architectural style are available from [wwwu]. All components shown in the C&C model in Figure 5.53 are identified as layers. The layers are arranged such that every layer only communicates with the layer above and the layer below. Interestingly, the outgoing port from layer `FM` has connectors to the layer `PCM` above and to the layer `FD` below.

To integrate this style with our synthesis solution we add the partition of the com-

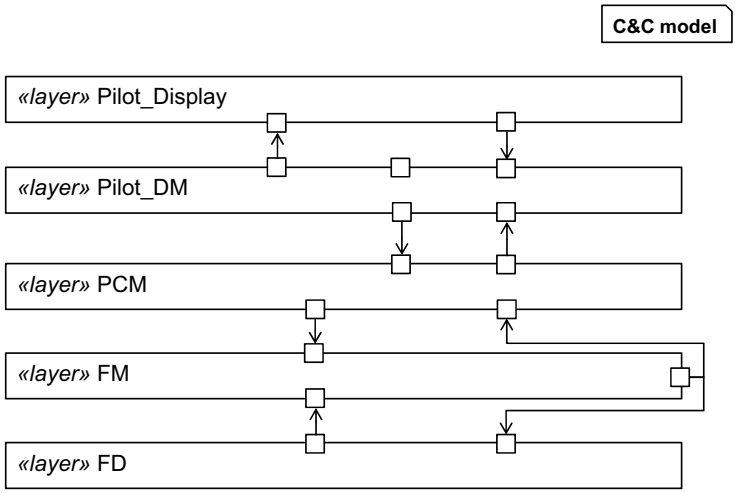


Figure 5.53.: A C&C model of the avionics system (see Section 5.6.2) in the layered architectural style. The complete C&C views specification is available from [wwwu].

ponents into layers, as defined by the architect, as additional input to the specification. In the example shown in Figure 5.53 the existing components are identified as layers. It is also possible to add new names of layers to the synthesis specification that are not contained in the set of components depicted in the views.

Definition 5.54 (Layer specification)**.** A layer specification $layers$, for a C&C views specification over the set of views $views$, is a list of pairs $(name, Cmps)$ where

- $name$ is the unique name of a layer and
- $Cmps \subseteq \bigcup_{v \in views} v.Cmps$ is the (incomplete) set of components contained in the layer $name$.

The name of every layer is unique and no two layers share components: $\forall layer_1, layer_2 \in layers : layer_1 \neq layer_2 \Rightarrow (layer_1.name \neq layer_2.name \wedge layer_1.Cmps \cap layer_2.Cmps = \varnothing)$. △

Definition 5.54 defines a layer specification that we add as an additional parameter for C&C views synthesis. For a layer $layer \in layers$ we interpret the set of contained components $layer.Cmps$ to be the minimal required set of components contained in the layer. Thus, we do not require that the assignment of components to layers is complete for all components in the views.

A layer specification $layers$ is a list of layer names and their contained components. Thus an engineer can specify the order of the layers in the synthesized C&C model. In the example shown in Figure 5.53 the first layer is the component `Pilot_Display`. All

of the following translation rules except for fact `orderOfLayers` of rule V7 shown in Figure 5.55 are independent of the given order of the layers and also support synthesis of layered C&C models without an order specified.

In every C&C model, satisfying the C&C views specification and the layer specification *layers*, the set of top components is the set of layers. Every layer $layer \in layers$ (transitively) contains at least the components $layer.Cmps$. The order of the layers in the specification is respected by the connectors of the C&C model: every layer in the list of layers has to be exclusively connected to its predecessor and successor, if it exists. The connection may be in one direction only, e.g., the layered C&C model from Figure 5.53 would still be valid if the connector from component `Pilot_Display` to component `Pilot_DM` would be removed since the two are still connected in the other direction.

To make the layers specification available in the Alloy module, we add the translation rule V7 defined in Figure 5.55 to the list of executed translation rules. The rule is executed at the same level as the rules V1 to V4 shown in the overview in Figure 5.18. The translation parameter *layers* contains the layer specification as defined in Definition 5.54. The generated Alloy function `myLayers` returns the Alloy atoms that represent the layer components. The fact `componentsInLayers` adds the containment constraints for the components contained in layers specified in the layer specification. The constraints are added using the Alloy predicate for transitive component containment shown in Listing 5.15. The fact `orderOfLayers` adds connectedness constraints between all layers in the list *layers* using the following notation: the first element of the list is $layers[1] = (\texttt{Pilot\_Display}, \varnothing)$ and $|layers|$ denotes the length of the list (number of elements). An example of the application of rule V7 is shown the lower part of Figure 5.55.

Translation Rule

Translation rule:

V7

```
fun myLayers : set Component {
  {layer.name}  +  layer∈layers
}

fact componentsInLayers {
∀layer ∈ layers :
  ∀cmp ∈ layer.Cmps :
    contains[layer.name , cmp.name ]
}

fact orderOfLayers {
  {immediatelyConnectedNoOrder[layers[i].name ,
                               layers[i + 1].name ]}  and  1≤i<|layers|
}
```

Result of application to a C&C views specification of the avionics system (see Section 5.6.2 and [wwwu] for the complete specification) with $layers = [(\texttt{Pilot\_Display}, \emptyset), (\texttt{Pilot\_DM}, \emptyset), (\texttt{PCM}, \emptyset), (\texttt{FM}, \emptyset), (\texttt{FD}, \emptyset)]$:

Alloy

```
1  fun myLayers : set Component {
2    FD + FM + PCM + Pilot_Display + Pilot_DM
3  }
4
5  fact componentsInLayers {
6
7  }
8
9  fact orderOfLayers {
10   immediatelyConnectedNoOrder[Pilot_Display, Pilot_DM] and
11   immediatelyConnectedNoOrder[Pilot_DM, PCM] and
12   immediatelyConnectedNoOrder[PCM, FM] and
13   immediatelyConnectedNoOrder[FM, FD]
14 }
```

Figure 5.55.: Translation rule V7 to create functions that return the atoms of components identified as layers in the layered style.

As in the client-server style, we replace the constraint of a single top component with a constraint that specifies that the layers are the top components. This is constraint is formalized in translation rule $V1_l$ which replaces the translation rule V1 defined in Figure 5.19.

Translation Rule

Translation rule:

$V1_l$

```
fact onlyLayersAreRoot {
   Component.subComponents = Component - myLayers
}
```

Figure 5.56.: Modification of translation rule V1 to $V1_l$ for the layered style. The fact states that all components except the layers are subcomponents.

We also adapt the translation rule V2 defined in Figure 5.20, which adds Alloy signature definitions for all components from the C&C views. In addition to the components from the C&C views, the translation rule $V2_l$ shown in Figure 5.57 also adds signatures for the layers defined in the layer specification (second part of rule $V2_l$). The difference between the signatures is that layers are forced to exist in any C&C model by using Alloy's multiplicity constraint `one` while the signatures for all other components are declared using the multiplicity `lone` as done in rule V2.

An application of the translation rule $V2_l$ to the avionics system with a layer specification is shown in the lower part of Figure 5.57. All components shown in the views of the specification are listed as layers in the layer specification.

Finally, we add a fact about the layered architectural style for C&C models shown in Listing 5.58 to the generic part of the translation. The fact `layeredArchitecture` specifies that every C&C model satisfying the layered architectural style has two outer layers and the set of layers specified is the union of the outer and inner layers. Outer layers are defined in the body of the function `outerLayers` in Listing 5.58, ll. 7-10, i.e., the layers that are directly connected to exactly one other layer (l. 8). In the avionics system C&C model shown in Figure 5.53 these are the top and bottom layers `Pilot_Display` and `FD`. Inner layers are defined in the body of the function `innerLayers` (lines 12-17), i.e., the layers that are directly connected to exactly two other layers (l. 16). Direct connectedness is specified by the predicate `immediatelyConnectedNoOrder` (ll. 19-23). Two components are directly connected if the first has a port that is either in the `receivingPorts` or in the `sendingPort` relation with any port of the second component.

If a C&C model that satisfies the specification together with the additional restrictions induced by the partition of the layers exists, it is found. If not, it means that the partition into layers conflicts with the semantics of the specification (or that the specification, even

Translation Rule

Translation rule:

$V2_l$ // components used in views that are not layers

$\forall name \in \{cmp.name \mid cmp \in \bigcup_{v \in views} v.Cmps\} \setminus \{l.name \mid l \in layers\}$:

lone sig *name* extends Component {}

// all layers of the C&C model

$\forall name \in \{l.name \mid l \in layers\}$:

one sig *name* extends Component {}

Result of application to a C&C views specification of the avionics system (see Section 5.6.2 and [wwwu] for the complete specification) with *layers* = [(Pilot_Display, ∅), (Pilot_DM, ∅), (PCM, ∅), (FM, ∅), (FD, ∅)]:

Alloy

```
1 // all layers of the C&C model
2 one sig FD extends Component {}
3 one sig FM extends Component {}
4 one sig PCM extends Component {}
5 one sig Pilot_Display extends Component {}
6 one sig Pilot_DM extends Component {}
```

Figure 5.57.: Translation rule $V2_l$ for components. Components identified as layers are required to exist in any synthesized C&C model.

without the style constraints, is not satisfiable, at least within the given scope).

5.5.4. On Styles and Views

It is important to note that each of the views in the specification is independent of and does not have to be compliant, by itself, with the constraints induced by the architectural style. For example, even though a layered architecture is enforced, abstract connectors in a view may connect components from nonconsecutive layers. The synthesis is responsible to implement these abstract connectors through chains of concrete connectors that obey the layered architecture. As another example, what looks like a communication cycle in a given view (and thus apparently violates a hierarchical style), may end up implemented in the synthesized C&C model without creating a cycle. For example, recall specification S_1 from Section 5.1.1 where view RJStructure shown in Figure 5.1 seems to contain a cycle between Body and Cylinder. In the synthesized C&C model, shown in Figure 5.4, we see that the implementation contains no concrete end-to-end cycle. Thus, the views and the architectural style are specified independently. The synthesis is responsible for finding a C&C model that satisfies both, if one exists.

Alloy

```
fact layeredArchitecture {
  # outerLayers = 2 and
  myLayers = innerLayers + outerLayers
}

fun outerLayers : set Component {
  {endP : myLayers |
    one partner : (myLayers - endP) |
      immediatelyConnectedNoOrder[partner, endP] }
}

fun innerLayers : set Component {
  {layer : myLayers|
    # {partner : (myLayers - layer) |
      immediatelyConnectedNoOrder[partner, layer]}
        = 2}
}

pred immediatelyConnectedNoOrder[c1: Component, c2: Component] {
  some p : c1.ports |
    ( p in c2.ports.receivingPorts or
      p in c2.ports.sendingPort )
}
```

Listing 5.58: Additional Alloy functions and predicates for the layered style. `myLayers` is a generated Alloy function returning the layer components (see translation rule V7 in Figure 5.55).

5.6. Implementation and Evaluation

The plug-in implementation and all specifications reported on below are available in supporting materials [wwwu], together with screen captures and relevant documentation. All specifications can be inspected and all experiments can be reproduced. We encourage the interested reader to try it out.

5.6.1. C&C Views Synthesis Plug-In

We implemented C&C views synthesis in a prototype Eclipse plug-in. The input consists of a C&C views specification. The specification — selection of views and components, definition of the propositional formula, scope, and optional parameters about styles — is edited using a user-friendly dedicated UI shown in Figure 5.59. At the back end, the plug-in implements the translation into Alloy using MontiCore APIs [KRV10] and FreeMarker [wwwf]. The SAT solver we use is MiniSat [wwwp]. When a C&C model is synthesized, it is presented to the engineer who can inspect it and further use it for code generation.

Our implementation consists of a library providing utility operations on C&C views and C&C models and a bridge from and to MontiCore ASTs, a synthesis engine and an Eclipse plug-in both written in Java. We report the size of the implementation in effective lines of code (ELOC). Lines counted as ELOC contain characters other than white space or comments and are contained in classes of the implementation. Thus the numbers of ELOC do not include unit tests and code for validation. The C&C views and C&C model utility library consists of 9 classes with a total of 911 ELOC (also used for the implementation reported on in Chapter 4). The C&C views synthesis engine consists of 14 classes with a total of 1,086 ELOC and 5 Freemarker templates with 368 ELOC. The C&C views synthesis plug-in consists of 38 classes with a total of 2,061 ELOC.

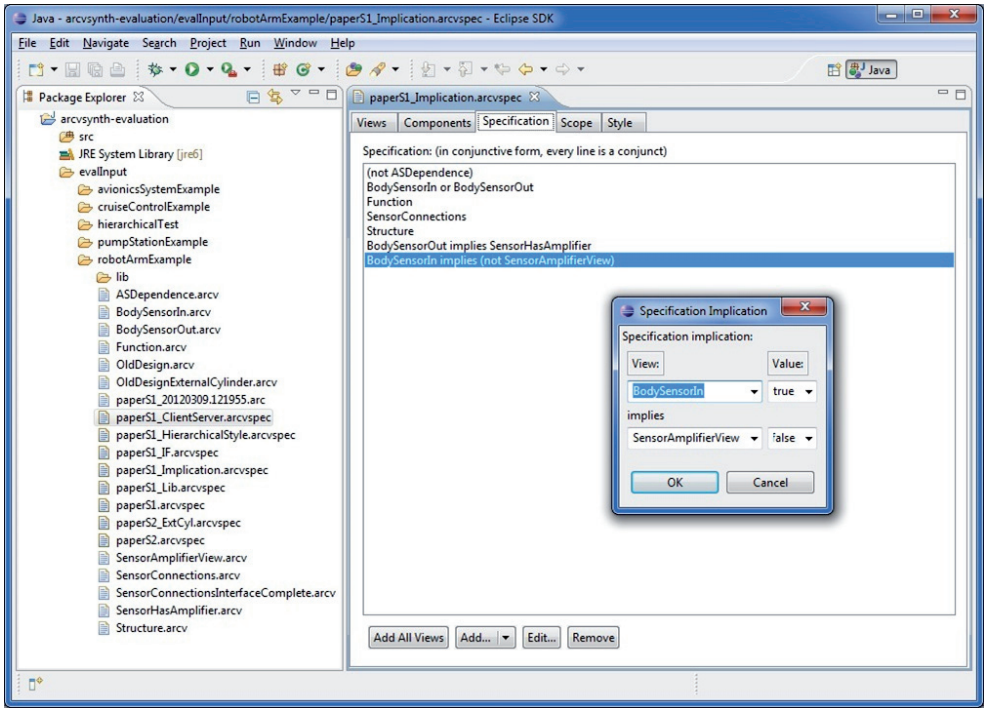


Figure 5.59.: A screen capture of the Eclipse plug-in for C&C views synthesis showing the specification editor.

We tested the implementation over C&C views specifications for four systems from different sources and of different domains (see Section 5.6.2). We experimented with several different specifications for each system, in order to test and evaluate the use of different features (e.g., library components, specification patterns, architectural styles). For validation, we have also applied our algorithm from Chapter 4 to verify whether a given C&C model satisfies a given view.

We present a tutorial on how to install the plug-in, import the example systems, edit C&C views specifications, and synthesize C&C models in Appendix D.

5.6.2. Evaluation Example Systems

We evaluated C&C views synthesis on four systems, taken from different sources. The evaluation based on example systems is of qualitative nature. We wanted to gain experience with creating C&C views for specifying C&C models and their use for synthesis. We discuss the results of our evaluation in Section 5.7.

Avionics system

We evaluated C&C views synthesis on an AADL C&C model of an avionics system, taken from [wwwa] (specifically `Avionics_System.aadl` of the OSATE AADL Project). The avionics system C&C model is a high-level model of several avionics system subsystems.

Based on various use cases related to interactions between system's components, we created 9 C&C views, 1-6 components each. For example, one view gives an overview of the complete data flow in the system, declared using abstract connectors. This view does not provide additional information such as port names or types. Another view provides more details about the communication between the `Pilot_Display` and it's `Page_Content_Manager`, showing incoming and outgoing ports with their names and connectors. We defined 7 satisfiable and unsatisfiable C&C views specifications, using 3 to 9 views, some extended with styles.

Pump station

We further experimented with a pump station design taken from an example system provided with the AutoFOCUS tool [HF07, wwwe]. The physical pump station system consists of two water tanks connected by a pipeline system with a valve and a pump. The water level in the first water tank can rise (this is controlled by the environment). When the water level of the first tank rises to a critical level, the water has to be pumped to the second water tank. The second water tank has a drain.

Based on several design decisions and relations we wanted to highlight and document, we created 10 C&C views, each with 2-5 components. For example, one view gives an overview of the basic structure of the system and omits details about interfaces and connectors. Another view documents part of the connections between the actuators and their environment, hiding hierarchies and omitting elements not connected to the actuators. An additional C&C view shows an undesired design where the simulation component is placed inside the pumping system.

We defined 8 satisfiable and unsatisfiable C&C views specifications. Two specifications specify the optional existence of an emergency system and its implications, using the [ALT] and [IMP] patterns. Another specification prohibits an emergency system. Other

specifications combine models of the function of the pump station with ones that specify the separation of the pumping system from the simulation part.

Robotic arm

We evaluated C&C views synthesis on a robotic arm C&C model – specifically a rotational joint, taken from an industrial system by VTT Tampere, Finland (the system used as running example in this chapter). The main components of the rotational joint's model are a cylinder, a servo valve, a sensor, a joint limiter, and an actuator. The rotational joint is a subsystem of a robotic arm containing eight rotational (identical copies) and translational joints in total.

Based on several requirements and partial knowledge or particular features, we created 11 C&C views, each with 1-5 components. Some views highlight the components necessary for the function of the joint while others document design alternatives on the placement of sensor and actuator components. Some of the views give an overview of related components with only few details of their interfaces or connectedness. Other views document complete interfaces of relevant components and some of their connections. Moreover, we created 8 C&C views specifications, each combining 6-8 C&C views to express design alternatives (pattern [ALT]), undesired designs, and implications of design decisions (pattern [IMP]).

Lunar lander

We evaluated C&C views synthesis on the lunar lander model, which is used by Taylor et al. as a running example in their book on software architecture [TMD09] and presented in a related work by Bagheri and Sullivan [BS10]. The lunar lander is a space ship with various sensors, a controller, and actuators. The objective of the lunar lander is to land safely on the surface of the moon.

Based on the natural language description of the lunar lander consisting of three components presented in [TMD09, pp. 201] we have created 8 C&C views, each with 1-3 components. Each C&C view covers parts of the natural language description. Since the natural language description is formulated positively, the C&C views specification is a conjunction of the 8 views.

Based on a shorter natural language description of the same lunar lander with 8 components (4 sensors, one controller, and 3 actuators) from [BS10] we have created 2 views and again the C&C views specification is a conjunction of these 2 views. Both original examples from [TMD09] and [BS10] contained neither port names nor types, and no hierarchy. We introduced port names in the C&C views of the lunar lander example from [TMD09].

Threats to validity

The choice of example systems for our experiments is limited to four systems from different sources. To address this threat, we have selected example systems from different

domains: avionics, automation, robotics, and control. Further studies with more example systems could provide more insight.

For three of the example systems only C&C models were available and we created the C&C views ourselves. An evaluation with independent subjects to assess the expressiveness of C&C views could address the threat of a possible bias. For the Lunar Lander example we derived the views from natural language descriptions of the architecture of the system. These descriptions only contain statements that result in positive views. More natural language descriptions of example systems are required to investigate this phenomenon.

5.6.3. Synthesis Results and Times

We have run performance experiments with different C&C views specifications to address the research question whether C&C views synthesis is feasible for the example systems. We were also interested in answering whether parameters of synthesis — including satisfiability, advanced features, and scopes — influence the performance of our synthesis implementation.

Table 5.60 summarizes the results of our experiments in synthesizing C&C models from different specifications for the four systems listed above. We show the concrete performance results in wall-clock time in order to give a general, rough idea about feasibility.

For each specification we report on the total number of components, number of views, the use of advanced features (library components, patterns, styles), the scope (upper bound for total number of ports) used, whether the specification was satisfiable or not, the components' nesting depth in the synthesized C&C model (if any), and the running times (in seconds). We discuss the performance results in Section 5.7.2.

We performed the experiments shown in Table 5.60 on a regular desktop computer, Intel Quad Core CPU, 3.4 GHz, running 64-bit Windows 7. The SAT solver we used is MiniSat [wwwp].

We have executed C&C views synthesis for the two specifications `LL-BS10` and `LL-TMD09` of the lunar lander with increasing scopes. For each of the specifications we started from the smallest scope that makes the specification satisfiable. In each run we increased the scope by one and report on the times it takes Alloy to translate the generated Alloy module into a propositional formula in conjunctive normal form (CNF). The CNF formula is subsequently solved by a SAT solver. We also report on the times it takes the SAT solver MiniSat to solve the formula for a C&C model that satisfies the specification. The results of the experiment are shown in Figure 5.61 for the specification `LL-BS10` with a resulting C&C model with 8 components and in Figure 5.62 for the specification `LL-TMD09` with a resulting C&C model with 3 components. The chart shown in Figure 5.61 contains a gap at scope 18. The actual time by the SAT solver to solve the CNF formula was more than five minutes (confirmed by multiple executions). We have omitted this number from the diagram to keep a scale better suited for all other results.

Spec Name	# Cmp.	# Views	Adv. Features	Scope	Sat.	Depth	Time (s)
RA: S_1	9	6	–	18	Y	3	9
RA: S_1IC	9	6	Int. complete	18	Y	3	62
RA: S_1LC	9	7	Lib. component	20	Y	3	7
RA: S_1IMP	9	8	Imp. pattern	18	Y	3	40
RA: S_1CS	9	6	CS arch. style	18	N	–	2
RA: S_1HIER	9	6	Hier. arch. style	18	Y	3	29
RA: S_2	9	7	–	18	N	–	2
RA: S_2XCYL	9	7	–	18	N	–	2
PS: ALL	16	6	–	30	Y	3	56
PS: ALL-EMRG	16	7	–	30	N	–	8
PS: ALL-EMRG-FIX	16	7	–	30	Y	3	59
PS: PHIS-SIM	16	4	–	20	N	–	2
PS: PHIS-SIM2	16	3	–	20	Y	3	6
PS: PHIS-SIM2-BADS	16	3	–	10	N	–	<1
PS: PHIS-SIM2-NO-EMRG	16	4	–	20	Y	3	16
PS: SENSOR-LIB	16	6	Lib. component	20	Y	3	2
AS: ALL	7	9	–	16	Y	2	2
AS: ALL-BADS	7	9	–	10	N	–	<1
AS: ALL-HIER	7	9	Hier. arch. style	16	Y	2	7
AS: ALL-LAY	7	8	Lay. arch. style	16	Y	2	1
AS: ALL-NOLAY	7	9	Lay. arch. style	16	N	–	<1
AS: PILOT	7	6	–	16	Y	2	<1
AS: PILOT-CS	7	3	CS arch. style	16	Y	2	<1
LL: LL-BS10	8	2	atomic comp.	14	Y	1	2
LL: LL-TMD09	3	6	–	22	Y	1	3

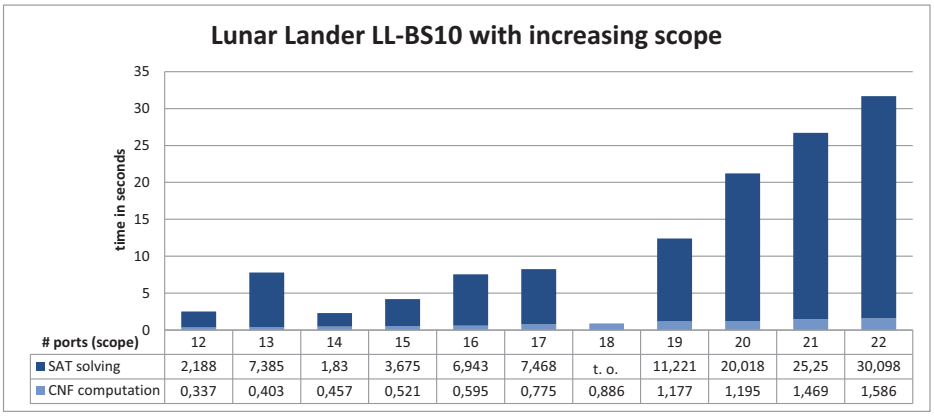
Table 5.60.: Results from running synthesis on several C&C views specifications for the robotic arm (RA), the pump station (PS), the avionics system (AS), and the lunar lander (LL). For each specification we report on the total number of components, number of views, the use of advanced features, the scope used (upper bound for total number of ports), whether the specification was satisfiable or not, the depth of the synthesized C&C model (if any), and the running times (in seconds).

We performed the experiments shown in Figure 5.61 and Figure 5.62 on a regular laptop computer, Intel Dual Core CPU, 2.8 GHz, running 64-bit Windows 7 and Java 1.7.0_17. The SAT solver we used is MiniSat [wwwp].

Threats to validity

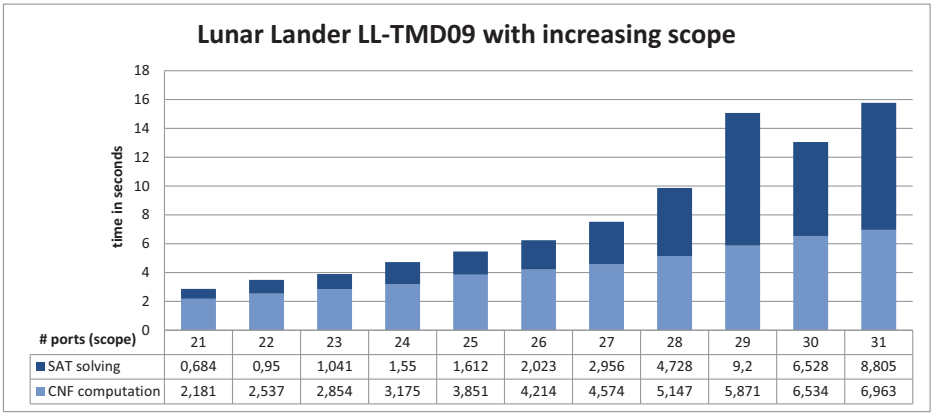
The choice of example systems used in our evaluation may not be representative of C&C models used in practice. To mitigate this we have chosen C&C models from different sources and domains.

We measured and reported the combined analysis times of our plug-in, Alloy, and the SAT solver in Table 5.60. Different versions of Alloy may include different internal translations to CNF which affect the running times of the SAT solver. Different SAT solvers and different SAT solver configurations may again affect analysis times. We have chosen the the SAT solver MiniSAT, which is widely used and readily integrated in Alloy with its default configuration.



# ports (scope)	12	13	14	15	16	17	18	19	20	21	22
SAT solving	2,188	7,385	1,83	3,675	6,943	7,468	t. o.	11,221	20,018	25,25	30,098
CNF computation	0,337	0,403	0,457	0,521	0,595	0,775	0,886	1,177	1,195	1,469	1,586

Figure 5.61.: Running times for synthesizing the lunar lander specification `LL-BS10` with increasing scopes for the number of ports. The scope for port names was set to 6 on all runs. We report times starting from scope 12, which is the first scope to make the problem satisfiable. All times for CNF computation and SAT solving are reported in seconds. For scope 18 the SAT solver timed out (t. o.) after 5 minutes.



# ports (scope)	21	22	23	24	25	26	27	28	29	30	31
SAT solving	0,684	0,95	1,041	1,55	1,612	2,023	2,956	4,728	9,2	6,528	8,805
CNF computation	2,181	2,537	2,854	3,175	3,851	4,214	4,574	5,147	5,871	6,534	6,963

Figure 5.62.: Running times for synthesizing the lunar lander specification `LL-TMD09` with increasing scopes. Starting from scope 21, which is the first scope to make the problem satisfiable. All times for CNF computation and SAT solving are reported in seconds.

Our experiments for synthesis of the same satisfiable C&C views specifications with different scopes was restricted to two example specifications. This impedes the generalizability of the findings that the scope influences synthesis times in an irregular way as shown in Figure 5.61. In our evaluation we focus however on the existence of the

phenomenon and not its generalization.

5.7. Discussion

In this section we report on the lessons learned from our evaluation and discuss various topics and advanced features of C&C views synthesis. Many of the topics, such as completeness and performance, are related to the choice of Alloy as a target formalism to express the C&C views synthesis problem and to using the Alloy Analyzer to find Alloy instances.

5.7.1. Completeness and Synthesis Scopes

Our solution for C&C views synthesis is based on a reduction to Alloy. Alloy is a model finder, which is complete in a bounded scope. The scope for the Alloy Analyzer is defined as either upper bounds or exact numbers of the atoms in each signature. The Alloy Analyzer is guaranteed to find a solution in the specified scope, i.e., it is complete. If a solution does not exist in a given scope a solution might exist for a different definition of the scope.

Since we use Alloy, the completeness of our synthesis solution also depends on the scope specified. An upper bound for the number of components in the C&C model is the number of components with distinct names in the C&C views. The upper bound for the number of port types is the number of different port types in all the views plus one (the additional type is required in case an untyped port is required to exist but all other types appearing in views are forbidden by the specification).

Remaining scopes to be given are the upper bound for the number of ports in the C&C model and the upper bound for the number of port names. One upper bound for the number of ports is the number of all abstract connectors in all views times two ports times the number of components in all views. This scope is large enough for every abstract connector to be implemented by an independent chain of connectors in the C&C model. An example of a C&C view that requires exactly this upper bound is shown in Figure 5.63. The view contains the two independent components `Body` and `Joint`. The component `Body` has n input ports each with a unique name. The component `Joint` has n output ports with the same n unique names. The C&C view also contains n abstract connectors connecting the input ports of the component `Body` to the output ports of the component `Joint`. To satisfy this view the chains of connectors need to be independent since their sources (the ports of component `Body`) are independent and every port in a C&C model may have only up to one incoming connector. Every chain of connectors requires one output port on the component `Body` and one additional input port on the component `Joint` not shown in the C&C view.

For most C&C views synthesis problems this upper bound is however computationally not feasible to use with our reduction to Alloy. For example, the lunar lander system `LL-BS10` from Section 5.6.2 has 7 abstract connectors and 9 components shown in all views. The synthesis problem with an upper bound of $2 * 7 * 9 = 126$ for the number of

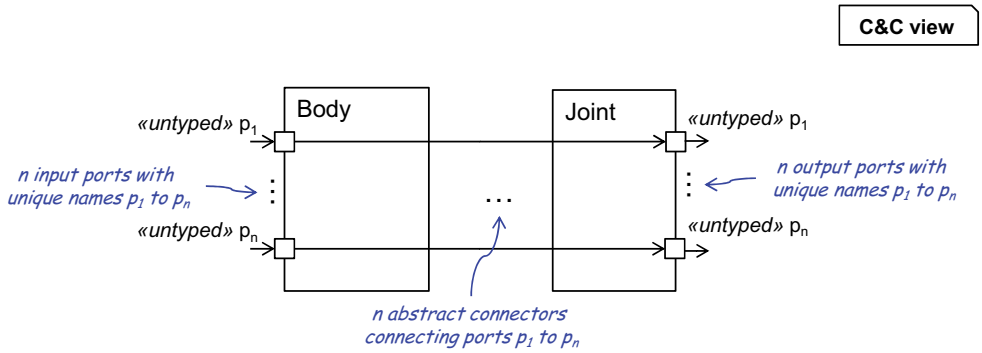


Figure 5.63.: A C&C view *view* that has only satisfying C&C models with at least $2 * |view.AbsCons| * |view.Cmps|$ ports. Every satisfying C&C model needs at least twice the number of port names shown in this view.

ports is not feasible to compute — as shown by the trend of the synthesis times shown in Figure 5.61. Thus, we currently leave it to the user to specify an upper bound for the number of ports in the C&C model to be synthesized. It might be possible to compute smaller upper bounds when taking the structure of the synthesis problems' views and propositional formula into account.

The number of port names used for synthesis again influences satisfiability. If the number of port names is chosen too low, synthesis might fail only because every port of a component needs a unique name. The number of port names is also the maximal number of ports per component. For n unique port names in all views our current implementation sets the upper bound for the number of port names to n if $n > 6$, otherwise it is set to the current default scope 6. Please note that the number 6 is chosen arbitrarily and should be replaced by a user defined number. In the example C&C view in Figure 5.63 the minimal number of ports required for synthesizing a C&C model is $2 * n$ where n is the number of ports on the component `Body` (same as for the component `Joint`).

5.7.2. Performance

For some specifications, synthesis took only a few seconds, while for others it took up to a minute to complete (on a regular desktop computer). Our experience shows that relatively minor changes in a specification, such as ones that add no views or components but only further constrain the specification with library components or statements of interface-completeness, and even ones that do not affect the semantics (such as different ordering of the views in the propositional formula), sometimes have a significant effect on performance. Indeed, it is known that the performance of Alloy and SAT solvers in general, is sensitive to the order of variables in the input.

Our definition of the C&C metamodel inside Alloy (see Section 5.3.2) contains some redundancies we added for convenience and readability. Examples of these redundancies are the relation `Component.parent`, which is the inverse of the relation `Component.subComponents`, and the relation `Port.sendingPort`, which is the inverse of the relation `Port.receivingPorts`. It is not clear whether removing these redundancies improves performance.

One observation is that synthesis times depend to a great extent on the scope (number of ports) chosen by the user of the plug-in. From the change in synthesis times with increasing scope shown in Figure 5.61 we can see that an increased scope significantly increases synthesis times. This is not surprising, since the time complexity of SAT solvers is exponential in their input. However it is also interesting to note that even for small scopes there are some outliers, e.g., scopes 13 and 18. The scope for C&C views synthesis is chosen by the user. Thus these outliers are interesting since they influence synthesis times in a way that is hard to predict. We leave it to future work to further investigate this issue.

Another observation from the two experiments is that for some specifications the time consuming part of synthesis is spent by Alloy's translation of the module to a CNF formula. From the times reported in Figure 5.62 we can see that the translation into a CNF formula from scope 21 to 29 was more time consuming than solving the formula. This suggests that a direct translation into a SAT problem instead of an intermediate translation into Alloy might be a possibility to speed up synthesis.

Since C&C views synthesis is NP-hard, one cannot expect it to be instantaneous. Thus, we consider the resulting times to be reasonable. Also, interestingly, deciding that a specification is not satisfiable was typically (but not always) much faster than deciding that it is satisfiable and providing the synthesized C&C model (see results reported in Table 5.60).

5.7.3. Language Expressiveness

We found that the use of a propositional formula over views, together with the patterns, in particular the use of alternatives, negations, and implications, is both expressive and easy to read and write. We thus believe that C&C views' 'by example' characteristics is attractive to engineers (our belief is supported also by the analogous use of scenarios as views in behavioral specifications). On the other hand, the use of views to specify some properties was not always natural and intuitive. Instead, sometimes we wished to have a more fine-grained, flexible, and powerful language that allows one to write symbolic, succinct specifications (e.g., using quantification). We leave this topic for future work.

5.7.4. Multiple Solutions

A C&C views specification may have more than one satisfying C&C model. For example, consider the specification S_1 of Section 5.1.1. Figure 5.4 shows a possible solution. An alternative satisfying C&C model may be a C&C model which is identical to the first except that `ServoValve` contains `Sensor`. If this is not an acceptable C&C model,

the architect can disallow it (e.g., by adding it, or a smaller view consisting of the `ServoValve` and the `Sensor`, in a negated form to the specification), and run synthesis again.

Most of our specifications had many satisfying solutions. So, we found that it may be useful to have a better way of choosing between solutions, e.g., by optimizing some cost functions (hierarchy depth, ports' number, connector chains length). This however cannot be efficiently done using our current technique. We leave this issue for future work.

5.7.5. Expressiveness vs. Performance

Our work uses a rather strong definition of a specification, supporting arbitrary propositional formulas over views. Alternatively, one may suggest to allow only a conjunction of views, or a conjunction of disjunctions of views, but without negation. Such a limited definition may lead to better performance by creating opportunities for the use of abstractions.

For example, we may consider abstracting ports and connectors away, dealing at first only with the subcomponent relation. Such an abstraction may be useful as a means to reduce the size of the problem and accelerate the computation, specifically for the purpose of falsification. If negation is not allowed, this is indeed an abstraction; that is, it will allow the same or more solutions (if a solution is not found, we know the original problem, without abstraction, has no solution too; if a solution is found, we need to go back to the original problem). However, if negation is allowed, ignoring ports and connectors may not only add but also eliminate possible solutions. Thus, this abstraction is possible only in case of specifications without negation.

On the other hand, a language without negation would indeed limit the expressive power of the views specification, e.g., without negation, one cannot specify exclusive alternatives or implications between views (see Section 5.4.3).

In future work it may be worthwhile to further investigate the trade-off between expressiveness and performance in the context of C&C views synthesis, in particular with regard to the use of abstractions on real-world views specifications.

5.7.6. Synthesis and Evolution

Software systems and their architectures evolve over time. Indeed, support for architectural evolution and the evolvability of C&C models (the ability to easily accommodate future changes) has been extensively studied (see, e.g., [MRT99, MMR10, BCL12]). Thus, in our context, one may be interested in an incremental variant of the synthesis problem where the input consists not only of a set of partial views but also of an existing C&C model.

The revised synthesis problem would be to compute an extension of an existing C&C model that satisfies a new views specification. The architect would specify which parts or properties of the existing C&C model may change and which ones may not. Please note that synthesis might fail if either the specification is unsatisfiable or an invariant

part of the model contradicts the updated views specification. It is possible to extend our current work to support this variant of the problem. We leave its formal definition, implementation, and evaluation to future work.

5.7.7. Handling Unsatisfiable Specifications

A C&C views specification may be unsatisfiable. In addition to identifying unsatisfiability, one may be interested in presenting the root cause of conflicts to the architect, i.e., to identify a minimal subset of the specification (propositional formula) that is unsatisfiable. Please note that unsatisfiability may have more than one cause, for example, as explained earlier in subsection 5.1.2, the specification S_2 is unsatisfiable both because of a conflict in component containment and because of a port type mismatch.

Heuristics may be used to detect some simple patterns of unsatisfiability in linear or polynomial time. A complete but inefficient solution to find a minimal subset would require an exponential number of synthesis runs. We believe that providing tools to handling unsatisfiable specifications efficiently is an important direction for future work. One may be able to build these on top of existing technologies for UNSAT core, as supported by some SAT solvers. However, in order to be effective, the identified core must be lifted and presented to the engineer back using the abstractions defined by the views.

5.7.8. Choice of Alloy as the Target Formalism

Our choice of Alloy as the target formalism for analysis, on the way to SAT, was motivated by Alloy's expressive power, readability, and readily available automated analysis. From our performance experiments we have seen that in some cases much of the time for solving the synthesis problem is used by Alloy to translate the specification into a CNF formula for the SAT solver (see the example in Figure 5.62). Thus, an alternative for C&C views synthesis could have been a direct translation to SAT, which may be better from a performance point of view. On the other hand, Alloy analyzer does use various heuristics to optimize the translation to SAT.

As a future work one could investigate the performance of C&C views synthesis using different model finders. A possible candidate is the relational model finder Kodkod [TJ07] used as a back-end by Alloy. Kodkod models are created using an API. They consist of relations and their bounds given as sets of tuples. Kodkod provides less abstractions than Alloy, e.g., fields and inheritance between signatures, but allows more control over the lower and upper bounds for relations. Another possible tool to solve the synthesis problem via a reduction is FORMULA [JLB11]. FORMULA takes as input domain descriptions as algebraic data types and constraint logic programs. It can complete partial models using a combination of symbolic execution and satisfiability modulo theories (SMT) solving. Similar to a direct reduction to SAT the synthesis problem could also be reduced directly to an SMT solver such as Z3 [dMB08]. SMT solvers typically solve first order logic formulas over background theories. The formulation of the model

finding problem for C&C views synthesis in terms of background theories, e.g., arrays and arithmetic, might allow a more convenient formulation compared to SAT.

We believe that the various extensions, e.g., styles and library components, we have presented and implemented are a good reason for the use of Alloy. Especially the implementation of styles required not only to add but also to modify parts of the existing translation rules. The abstractions provided by Alloy make these extensions possible with reasonable effort. Our rule-based translation using concrete syntax templates with the readable syntax of Alloy has proven helpful for modifications and maintenance.

Finally, we consider our implementation using Alloy to be a sufficient proof of concept research prototype for C&C views synthesis. In the future it may be interesting to compare our implementation with ones that use alternative encodings and tools.

5.8. Related Work

Many works have suggested synthesis and composition methods related to architecture views. These include differencing and merging (e.g., Abi-Antoun et al. [AAAN+08], Chen et al. [CCG+03], Sabetzadeh and Easterbrook [SE04, SE06]), as well as related formal analyses of architectures within architecture styles (e.g., Kim and Garlan [KG10]). Boucké et al. [BWH10] present composition operators for architectural models. Their approach is imperative rather than declarative. Giese and Vilbig [GV06] discussed separation of non-orthogonal concerns in software architecture and design. The work deals with composition of structure as well as composition of behavior.

We review related works for views merging, synthesis approaches for behavior in the context of software architecture, and formalizations of architecture analyses based on Alloy below. To the best of our knowledge, no previous work has suggested to consider views as specifications for the structure of architectures and use them for an automated synthesis process, as is done in our work.

5.8.1. Views Merging

Chen et al. [CCG+03] present differencing and merging techniques for product-line architectures, specifically in the context of evolution. Differences are categorized as elements that have been added, elements that have been removed, or elements that have changed. The differences are represented in XML format using a schema that extends xADL [DvdHT01] and then merged into a target architecture.

Abi-Antoun et al. [AAAN+08] present differencing and merging of architectural views using a technique that detects renames, inserts, deletes, and restricted moves. The technique is inspired by tree-to-tree correction algorithms.

Sabetzadeh and Easterbrook introduce a graph based framework for view merging for arbitrary modeling languages [SE04, SE06]. Their framework is modeling language independent. The merging of multiple views integrates the information contained in several partial views. One weakness of the framework is that it does not handle complex well-formedness rules of the views merged. Examples for these rules in C&C views are

that no component contains itself. A merge of views using the framework from [SE06] might result in a structure that is neither a view nor a C&C model.

Boucké et al. [BWH10] present composition operators for C&C models to avoid the repetition of elements in integrated models. The integration of sets of architecture models is based on a user specified unification relation (marking identical elements across input models) and a submodel relation (e.g., a component is detailed in another model). This model composition is thus an imperative composition of concrete C&C models rather than a declarative one as in our approach. We believe that our work may benefit as well from a more complex and maybe user specified unification relation across C&C views as suggested in [BWH10].

5.8.2. Synthesis Related to Architecture Behavior

Issarny et al. [KI01, IKZ02] present a development environment for the composition of middleware architectures. The input for composition are two C&C models. The output of the directed composition operator is a single C&C model where possibly multiple copies of components of the second model are connected with components of the first. A valid composition preserves communication order of the input models and CTL properties checked against PROMELA implementations of components, both specified by the user. The work may be viewed as similar to ours, as it translates a C&C model composition problem into a model checking problem. It differs significantly, since the input in [KI01, IKZ02] consists of concrete models without common elements, without hierarchies (they are flat) or abstraction mechanisms as supported by C&C views. Moreover, their composition is binary and directed while ours supports propositional specifications over C&C views, including negation, alternatives etc.

Giese and Vilbig [GV06] discuss separation of non-orthogonal concerns in software architecture and design. The work deals with composition of structure and behavior from architectural views. Architectural views are defined as directed graphs representing components and connectors extended with behavior contracts. According to [GV06], the structural part of their architectural view synthesis can be handled by superposition of these directed graphs. In contrast, the synthesis problem for our C&C views specifications is more complex because of its rich abstraction mechanisms (of direct hierarchy and connectivity) and its support for the specification of negative or alternative designs, which prohibit simple superposition.

From a broader, high-level perspective, variants of synthesis from partial views have been studied in the area of behavioral specifications (see, e.g., the works of Harel et al., Uchitel et al., and Whittle et al. on synthesizing behavior models from scenarios [WS00, UBC07, HS12]). These works assume the structure of the systems to be given; only the behavior is synthesized. Our present work deals with the synthesis of structures, not behaviors, and may be viewed as complementing some of these works. As future work one may develop integrated structural and behavioral synthesis techniques.

5.8.3. Analyses of Architecture Models using Alloy

Jackson and Sullivan [JS00] analyze a part of the Microsoft Component Object Model (COM) [Box98] using the Alloy Analyzer. They have translated a Z [Spi92] specification of COM into a specification in the Alloy language. The resulting Alloy formalization contains hierarchical components with connected interfaces similar to our Alloy meta-model for C&C models. The goal of Jackson and Sullivan was to check properties of the specification and to refine the model to a simpler representation. The analysis was not on specific C&C models or C&C views as in our case but on the meta-model itself.

Kim and Garlan [KG10] present formal analysis of C&C models within architecture styles, focusing on structural properties. Structural architectural style definitions are translated into a relational model, specifically in Alloy [wwwb, Jac06], in order to allow various consistency checks and related analyses.

Bagheri and Sullivan [BS10] use Alloy to construct a style-specific architecture from a style-independent application model. Their work is based on the observation that an application can be described independently of an architectural style [BSS10]. Unlike our work, the input for their technique is a single style-independent application model, not a propositional formula over a set of partial views. As styles are common and successful design practices, we have included support for some architectural styles in our synthesis solution (see Section 5.5).

Bagheri and Sullivan [BS10] use the lunar lander example as a case study. Although this example is small and flat (and thus does not take advantage of the unique features of our approach), we use it in our evaluation too, for the purpose of comparison (see Section 5.6).

Chapter 6.

MontiArcAutomaton: State-Based Behavior Modeling

The software components in interactive systems communicate via messages sent and received through their ports. Examples of components are, e.g., sensors that produce data messages, controllers that monitor sensor data and issue actuator commands, and actuators that consume commands for actions to execute on a physical system. We are most interested in discrete controllers that control cyber-physical systems based on data read from sensors and the actions effected by actuators.

We use a state-based model for the interaction of components described in MontiArc software architectures. We implement a description technique in the new modeling language MontiArcAutomaton that extends MontiArc with automata following the I/O$^{\omega}$ automaton paradigm [Rum96]. Examples of the kind of systems we model are embedded software systems from the automotive domain or the software systems of robots.

As a system model for distributed interactive systems we employ streams, I/O relations, and stream processing functions from FOCUS [BS01]. These concepts form a semantic domain for MontiArcAutomaton models. We introduce a language profile for MontiArcAutomaton to model time-synchronous component behavior on which we focus in the remainder of this work.

The MontiArcAutomaton modeling language implements the underspecification mechanisms of I/O$^{\omega}$ automata: behavior descriptions of components may be non-deterministic and different completions complement the transition relation with behavior. Based on the FOCUS theory of behavior refinement, MontiArcAutomaton models thus serve as behavior specifications as well as implementations.

We discuss two semantics of MontiArcAutomaton automata: one based on stream processing functions from [Rum96] and one based on relations of input and output streams from [BS01]. We chose to present both semantics because the first is used for the semantics definition of I/O$^{\omega}$ automata on which MontiArcAutomaton automata are based and the second because it is commonly used in the FOCUS framework. In Section 6.5 we show that the two semantics definitions lead to different notions of refinement for the MontiArcAutomaton language profile we focus on. Highlighting this difference is important for the remainder: in Chapter 7 we rely on the I/O relation semantics with a conceptually simpler semantic domain than stream processing functions.

We have summarized the preliminaries given in Section 6.3 in [RR11] and presented the modeling language MontiArcAutomaton in [RRW12, RRW14].

Chapter outline and contributions

We present an example of a basic robotic system whose behavior can be described and implemented using automata of the modeling language MontiArcAutomaton in Section 6.1. Section 6.2 gives an overview of the MontiArcAutomaton modeling language as one contribution of our work.

Preliminary work on streams, I/O relations, and stream processing functions from Focus [BDD+92, Bro93, Rum96, BS01, RR11] is summarized in Section 6.3.

We introduce a language profile for MontiArcAutomaton in Section 6.4, on which we focus in the remainder of this work. As one of our contributions we present formal semantics for MontiArcAutomaton components. Section 6.5 shows that the semantics definitions based on stream processing functions and I/O relations lead to different notions of refinement for MontiArcAutomaton.

Finally, Section 6.6 surveys related modeling formalisms for state-based descriptions of interactive systems.

6.1. Example of a Reactive System

As an example for a reactive system consider the bumper bot robot shown in Figure 6.1. The robot has a touch sensor mounted at the front, a wheel in the back to balance, and two motors, one on each side of the robot powering a wheel. The objective of the bumper bot is to traverse an area by going straight forward until it hits an obstacle. The robot then backs up, turns around, and proceeds forward. The complete component and connector software architecture of the robot is shown in Figure 6.2.

Figure 6.1.: The bumper bot robot with a touch sensor in front and two motors to power the left and right wheels.

The components `TouchSensor`, `Timer`, and the parametrized components `Motor` are components from a Lego NXT specific component library and wrap access to hardware — touch sensor, clock, and motors of the robot. The state-based description of the

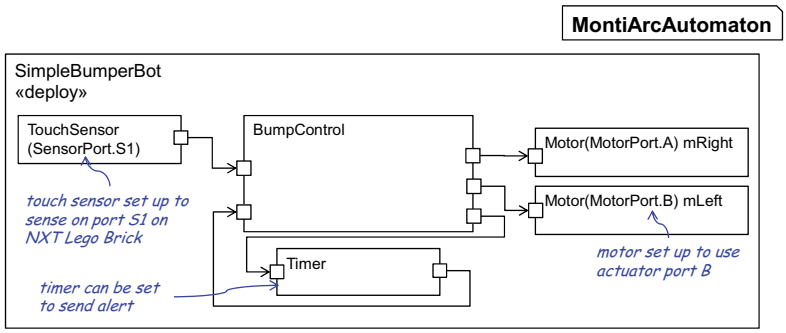


Figure 6.2.: The C&C architecture `SimpleBumperBot` of the bumper bot.

behavior of the component `BumpControl` is shown in Figure 6.3. The types shown on the ports of the component are the basic type `Boolean` = {**`true`**, **`false`**} and the enumeration types `TimerSignal`, `MotorCmd`, and `TimerCmd` defined in the class diagram shown in Figure 6.4.

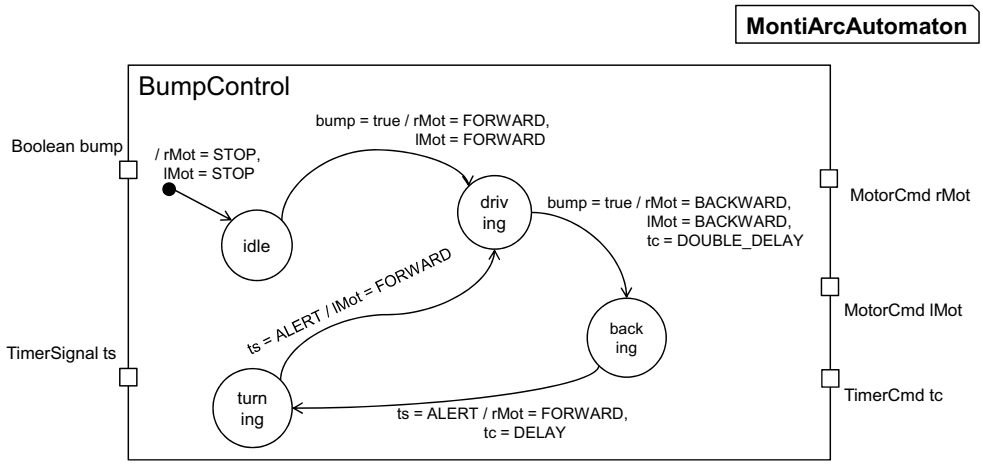


Figure 6.3.: The component definition and state-based behavior description of component `BumpControl` of the bumper bot.

The component `BumpControl` shown in Figure 6.3 initially stops the motors by sending a `STOP` command on both output ports `rMot` and `lMot` of the type `MotorCmd`. The component activates the driving mode of the robot once the bump sensor is pressed. It sends commands for both motors to drive forward and the controller switches to the state `driving`. In case the controller registers bumping into any object by receiving

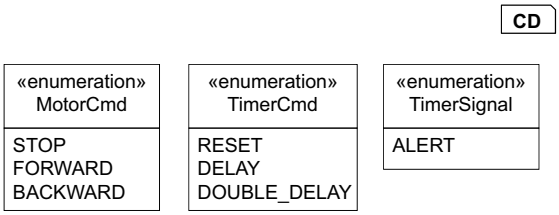


Figure 6.4.: A UML/P class diagram defining the enumeration types `TimerSignal`, `MotorCmd`, and `TimerCmd`.

the message `true` on port `bump`, both motors are set to drive backwards. In addition, a timer is set with the command `DOUBLE_DELAY` sent on port `tc`. The controller waits in the state `backing` until it receives the message `ALERT` on the port `ts`. The controller then issues motor command for the robot to turn around by going forward with one motor while the other one still rotates backwards. At the end of another delay the second motor is also set to rotate forward and the controller goes back to the state `driving`.

6.2. MontiArcAutomaton Modeling Language

MontiArcAutomaton is a modeling language for modeling the behavior of components in C&C models. We describe the behavior of components depending on their current state as the reaction to the inputs a component receives on its ports. The reaction of a component is expressed by the messages sent on the component's output ports. The internal states and processing steps of components are hidden from the outside. This implements the concept of information hiding of component-based design where only information required for the use of the component is made public through its interface. This follows the compositional approach of FOCUS implemented in MontiArc.

The MontiArcAutomaton modeling language is an implementation of the I/O$^{\omega}$ automata paradigm presented in [Rum96]. There are some differences between I/O$^{\omega}$ automata and MontiArcAutomaton. First, I/O$^{\omega}$ automata allow infinite output to describe non-terminating actions. We restrict the output per port and transition in MontiArcAutomaton to finite sequences. Second, I/O$^{\omega}$ automata allow an infinite state space. In MontiArcAutomaton states have to be enumerated and their number is thus finite. MontiArcAutomaton however allows an infinite state space, expressed using variables. Finally, MontiArcAutomaton extends I/O$^{\omega}$ automata with explicit support for multiple ports, support for guards on transitions, and support for local variables. These extensions are well-known from existing modeling languages [Har87, Obj12b, Obj12a, Rum11].

Listing 6.5 depicts a model expressed in the concrete syntax of the MontiArcAutomaton modeling language. The modeling language MontiArcAutomaton extends the modeling language MontiArc. Every MontiArc model is also a MontiArcAutomaton model. In addition to the concepts of MontiArc (see [HRR12]), the definition of a com-

ponent may contain variable declarations and automata. The component type definition `BumpControl` in Listing 6.5 shows a component with the input ports `bump` of type `Boolean` and `ts` of type `TimerSignal` and three output ports.

MontiArcAutomaton

```
1  package bumperbot;
2
3  component BumpControl {
4
5    port
6      in Boolean bump,
7      in TimerSignal ts,
8      out TimerCmd tc,
9      out MotorCmd rMot,
10     out MotorCmd lMot;
11
12   automaton {
13     state
14       idle, driving, backing, turning;
15     initial
16       idle / {rMot = STOP, lMot = STOP};
17
18     idle -> driving    {bump = true} /
19                        {rMot = FORWARD,
20                         lMot = FORWARD};
21     driving -> backing {true} /
22                        {rMot = BACKWARD,
23                         lMot = BACKWARD,
24                         tc = DOUBLE_DELAY};
25     backing -> turning {ts = ALERT} /
26                        {rMot = FORWARD,
27                         DELAY};
28     turning -> driving {ALERT} /
29                        {lMot = FORWARD};
30   }
31 }
```

Listing 6.5: The MontiArcAutomaton model of the component `BumpControl`.

The component definition of the component `BumpControl` contains an automaton (Listing 6.5, ll. 12-30). The automaton has four states `idle`, `driving`, `backing`, and `turning`. The block starting with the keyword `initial` (l. 15-16) declares the state `idle` as an initial state that sends the initial output `STOP` of type `MotorCmd` on the two outgoing ports `rMot` and `lMot`. The transitions in the example have a source state and a target state separated by `->`, an input block, and an output block. The first transition from the initial state `idle` goes to state `driving` if the input on port `bump` is the value **`true`** (ll. 18-20). When the transition is taken, the value `FORWARD` is sent

on the two outgoing ports `rMot` and `lMot`.

MontiArcAutomaton has some syntactic convenience mechanism. Instead of writing the port name and value, it is enough to write the port's value if the port name can be uniquely determined by the type of the value. Examples for this notation are given in line 21 and line 28 of Listing 6.5. Here, the value's type uniquely determines the name of the input port where the value is read. The same short notation also works for output ports and values as shown in line 27.

MontiArcAutomaton

```
1  package util;
2
3  component Buffer<T> {
4
5    port
6      in Boolean request,
7      in T data,
8      out T response;
9
10   T storage;
11
12   automaton {
13     state
14       buffering;
15     initial
16       buffering;
17
18     buffering {request = true}  / {response = storage,
19                                    storage = data};
20     buffering [ocl: data != null && request == false]
21                                 / {storage = data};
22   }
23 }
```

Listing 6.6: The MontiArcAutomaton model of the generic component `Buffer`.

As another example for the concrete syntax and features of the modeling language MontiArcAutomaton, consider the component `Buffer<T>` shown in Listing 6.6. The component has a generic data type `T` in its signature. When instantiating the component `Buffer<T>`, a concrete type, e.g., `String` or `Integer`, has to be supplied for the type variable `T`. The component `Buffer<T>` can thus be used to buffer messages of any type. In line 7 and line 8 of Listing 6.6 the generic type `T` is used as the data type of the input port `data` and the output port `response`.

MontiArcAutomaton allows the definition of variables inside components, e.g., to store data of previous calculations. An example of the concrete syntax for the declaration of variables is shown in line 10 of Listing 6.6 on the example of the declaration of a variable `storage` of the type `T`. The automaton implementing the behavior of the component

`Buffer<T>` has a single state `buffering`. The first transition of the automaton has the same source and target state `buffering` which can be written in the short form shown in line 18 (omitting the arrow and target state). In case the input port `request` receives the Boolean message **`true`**, the value stored in the local variable `storage` is sent on the port `response` and the value received on the input port `data` is stored to the variable `storage`. Please note that new values are assigned to variables only for the next execution cycle of the automaton, i.e., independent of the order of the assignments in lines 18 and 19 the previous value of the variable `storage` is sent on the output port `response`. The elements on the right side of assignments to ports and local variables in output blocks may be concrete values, local variables (denoting the value of the variable) or names of input ports as shown in line 19 (denoting the value received on the port).

The transition in Listing 6.6, l. 20 is guarded by an OCL/P [Rum11, Sch12] predicate to ensure that the value on the port `data` is not `null` and that no request is issued. Please note that we have defined a semantics for embedded OCL/P expressions in MontiArcAutomaton. In the semantics used in this example `data != null` means that *some meaningful* value is received on the input port `data` and `request == false` means that the value **`false`** is read on the input port `request`. Our code generator for the robotics platform leJOS, presented in Chapter 8, translates embedded OCL/P expressions in guards according to this semantics. The modeling language MontiArcAutomaton currently supports embedded OCL/P and embedded Java guards.

The output of each transition of MontiArcAutomaton automata can be specified for multiple output ports. The modeling language MontiArcAutomaton allows the output on each port to be a finite stream of messages. An example is the concatenation of the three messages `STOP`, `BACKWARD`, and `FORWARD` on the output port `rMot` as shown in line 19 of Listing 6.7.

In case multiple messages can trigger the same transition, MontiArcAutomaton allows to define these alternatives in one transition as shown in Listing 6.7, l. 20. More complex conditions may be expressed using guard expressions in an embedded guard language.

In this language overview we only presented the main features of the MontiArcAutomaton modeling language. Appendix J contains a complete grammar. A detailed description of the MontiArcAutomaton language is given in a technical report [RRW14] with a detailed introduction of the syntactic elements and a complete list of context conditions. We define a language profile of MontiArcAutomaton with a subset of the features mentioned here in Section 6.4 that we use throughout the remainder of this thesis.

6.2.1. Component Type Definitions

In many cases in component based system development the same component type is used multiple times in a software system. Examples are generic component types such as buffers, filters, or components wrapping access to hardware as the components `Touch-Sensor` and `Motor` from the initial example in Figure 6.2. To allow the reuse of components, the ADL MontiArc distinguishes between the definition of a component and the instantiation of a component.

MontiArcAutomaton

```
package robot;

component ManeuverController {

  port
    in HighLevelManeuver maneuver,
    out MotorCmd rMot,
    out MotorCmd lMot;

  automaton {
    state
      stopped,
      // ... more states ...
      driving;
    initial
      stopped;

    driving             {TURN_RIGHT} /
                        {rMot = STOP:BACKWARD:FORWARD};
    driving -> stopped {HALT | STOP} /
                        {rMot = STOP, lMot = STOP};
    // ... more transitions ...
  }
}
```

Listing 6.7: An excerpt of the automaton inside component `ManeuverController` in concrete MontiArcAutomaton syntax.

Previously, we have mainly considered C&C models for describing complex systems as instances of hierarchically composed components. We now introduce component type definitions in Definition 6.8. A component type definition can be seen as a blueprint, which describes the elements required to instantiate a component of the defined component type.

Definition 6.8 (Component type definition)**.** Given a universe of names $Name$, a universe of ports $Port$, and a universe of data types $Type$, a component type definition is a structure $cmp = (cType, CPorts, CVars, CSubCmps, CCons) \in CTDefs$ where

1. $cType \in Name$ is the unique type name of the component $(\forall c_1, c_2 \in CTDefs : c_1.cType = c_2.cType \Rightarrow c_1 = c_2)$,
2. $CPorts \subseteq Port \subseteq (dir : \{IN, OUT\} \times name : Name \times type : Type)$ is the set of directed input and output ports of the component type $cType$ where each port $p \in CPorts$ has a direction $p.dir$, name $p.name$, and a type $p.type$,
3. $CVars \subseteq (name : Name \times type : Type)$ is the set of local variables of the component type $cType$, where each $v \in CVars$ has a name $v.name$ and a type $v.type$,

4. $CSubCmps \subseteq (name : Name \times type : CTDefs)$ is the set of subcomponents of the component type $cType$ where each subcomponent $sub \in CSubCmps$ has a name $sub.name$ and a component type definition $sub.type$, and

5. $CCons \subseteq (srcCmp : Name \times srcPort : Port \times tgtCmp : Name \times tgtPort : Port)$ is a set of directed connectors $con \in Cons$, each of which connects two ports of the same type $con.srcPort.type = con.tgtPort.type$, with $con.srcPort \in getCTDef(con.srcCmp).CPorts$ and $con.tgtPort \in getCTDef(con.tgtCmp).CPorts$, where

   $$getCTDef : name \mapsto THE\ t \in CTDefs : (name, t) \in CSubCmps \cup \{(cType, cmp)\}$$

   retrieves the unique component type definition for the name $cType$ or the subcomponent name $name$.

Component types are either *atomic* where $CSubCmps = \varnothing = CCons$ or *composed* where $CSubCmps \neq \varnothing \vee CCons \neq \varnothing$ and $CVars = \varnothing$. The following rules for well-formedness apply:

6. $\forall c \in CTDefs : \nexists (n_1, c_1), .., (n_k, c_k) : (n_1, c_1) \in c.CSubCmps \wedge c_k = c \wedge \forall i < k : (n_{i+1}, c_{i+1}) \in c_i.CSubCmps$, i.e., no component type definition transitively contains itself and

7. $\forall pv_1, pv_2 \in CPorts \cup CVars : pv_1.name = pv_2.name \Rightarrow pv_1 = pv_2$, i.e., the names of ports and variables are unique within the component type definition.

Additionally, for composed component types:

8. $\forall sub \in CSubCmps : sub.name \neq cType \wedge \forall sub_1, sub_2 \in CSubCmps : sub_1.name = sub_2.name \Rightarrow sub_1 = sub_2$, i.e., each subcomponent has a unique name different from $cType$,

9. $\forall con_1, con_2 \in CCons : con_1.tgtCmp = con_2.tgtCmp \wedge con_1.tgtPort = con_2.tgtPort \Rightarrow con_1 = con_2$, i.e., each port has at most one incoming connector,

10. Each connector $con \in CCons$ satisfies exactly one of the four cases
    (a) $con.srcCmp = cType = con.tgtCmp \wedge con.srcPort.dir = IN \wedge con.tgtPort.dir = OUT$, i.e., the component directly forwards input as output,
    (b) $con.srcCmp = cType \wedge \exists sc \in CTDefs : (con.tgtCmp, sc) \in CSubCmps \wedge con.srcPort.dir = IN \wedge con.tgtPort = IN$, i.e., the parent component forwards input to a subcomponent,
    (c) $con.tgtCmp = cType \wedge \exists sc \in CTDefs : (con.srcCmp, sc) \in CSubCmps \wedge con.srcPort.dir = OUT \wedge con.tgtPort.dir = OUT$, i.e., the parent component forwards the output of a subcomponent, or
    (d) $\exists s_1, s_2 \in CTDefs : (con.srcCmp, s_1), (con.srcCmp, s_2) \in CSubCmps \wedge con.srcPort.dir = OUT \wedge con.tgtPort.dir = IN$, i.e., two subcomponents are connected or, for $s_1 = s_2$, a subcomponent receives its own output as input.

11. $\forall p \in CPorts : p.dir = OUT \Rightarrow \exists con \in CCons : con.tgtCmp = cType \wedge con.tgtPort = p.name$, i.e., all outgoing ports of the component are the target of an internal connector.

△

Notation: For a component type definition $c \in CTDefs$ we denote the set of input ports $\{p \in c.CPorts \mid p.dir = IN\}$ as $c.CPorts_{IN}$ (respectively $c.CPorts_{OUT}$ for output ports).

The structures of component type definitions (Definition 6.8) and C&C models (Definition 2.2) appear similar, since both describe components, ports, and connectors. They are however conceptually very different: a component type definition describes the structure of a single component and references its immediate subcomponents, while a C&C model describes a complete system composed of component instances.

An element not known from C&C models is the set of local variables $CVars$ of component types. In addition, the subcomponent relation is fundamentally different. Subcomponents in component type definitions have a name and a type. The name of a subcomponent is unique in the scope of the component type. The type of the subcomponent refers to another component type definition. In contrast, in C&C models the names of components are unique in the model and components do not have a type.

Connectors in component type definitions are again very different from connectors in C&C models. A connector in a component type definition connects either a parent component to its direct subcomponents or a subcomponent to its siblings. It is important, that the connectors identify components by their locally unique name inside the component type definition and not by their component type. A component type definition might have multiple subcomponents of the same component type.

A component type definition $c \in CTDefs$ may be instantiated to a C&C model m, where c defines the ports, connectors and subcomponents of the top level component $cmp \in m.Cmps$. The structure of the immediate children of the component cmp is defined by the component type definitions in the name and type pairs $c.CSubCmps \subseteq (name : Name \times type : CTDefs)$.

We repeat the illustration of component type instantiation previously shown in Figure 2.5. Figure 6.9 illustrates the instantiation of a component type definition to a corresponding C&C model. The component type definition `ToggleSensor` defines that components of the type `ToggleSensor` have two subcomponents: `sensor` of type `TouchSensor` and `switch` of type `ToggleSwitch`. Components of the type `TwoSwitchController` have three subcomponents: `tsens1` and `tsens2` both of type `ToggleSwitch` and `controller` of type `Controller`.

The referenced component types `TouchSensor`, `ToggleSwitch`, and `Controller` are atomic. Their component type definitions are given in the set *CTDefs* (see Definition 6.8, Item 4), but are not shown in Figure 6.9. Component type definitions of atomic components define neither subcomponents nor connectors. The component type definition `TwoSwitchController` is instantiated to the C&C model shown in Figure 6.9 (c).

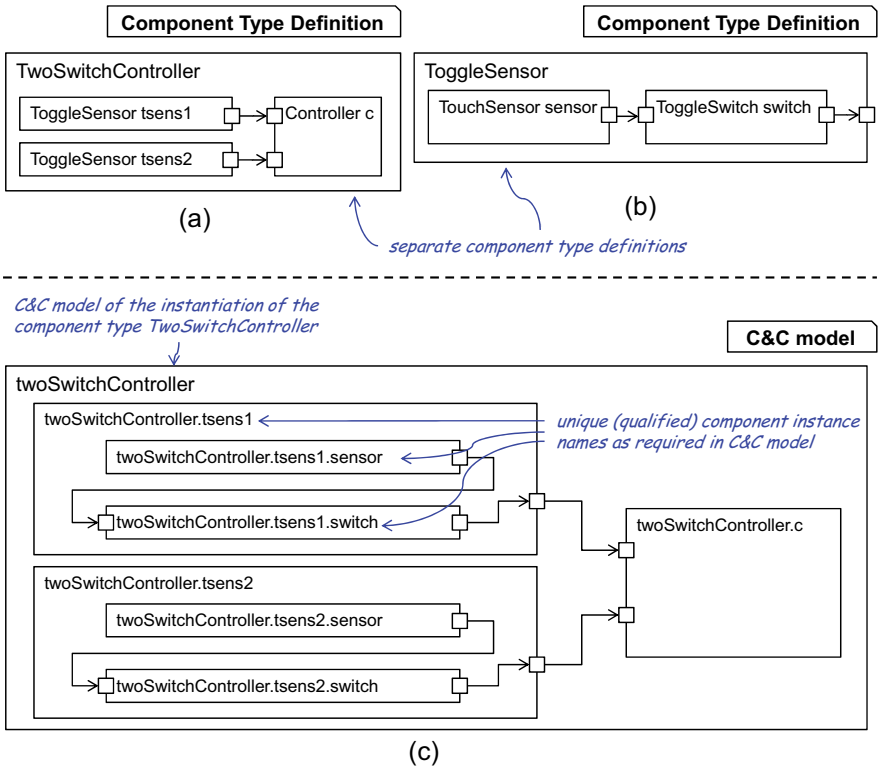


Figure 6.9.: Two component type definitions (upper part) and the instantiation of component type `TwoSwitchController` as a C&C model (lower part). The component type `TwoSwitchController` defines two subcomponents of the component type `ToggleSensor`. The referenced component types `TouchSensor`, `ToggleSwitch`, and `Controller` are atomic and not shown in the figure. The C&C model on the right is an instance of the component type `TwoSwitchController` that contains two instances of the component type `ToggleSensor`.

The well-formedness rules for component type definitions in Definition 6.8 ensure that the corresponding C&C model is also well-formed with respect to the rules in Definition 2.2.

In addition to the features listed in Definition 6.8, component type definitions of MontiArcAutomaton support parameters to configure component instances. An example shown in Figure 6.2 is the parameter value `MotorPort.A` of component type `Motor` that configures the component instance to use the hardware port `MotorPort.A` on the NXT controller. Component definitions also support type parameters to define, e.g., the type of ports in component definitions and instances as shown in the example of the

component `Buffer<T>` in Listing 6.6.

Definition 6.8 abstracts MontiArcAutomaton's component type name, the component parameters, and the type parameters to the single element `cType`. We omit the implementation details of these advanced concepts, which are not required for consecutive definitions and the techniques presented in Chapter 7. The complete structure of component type definitions is given in the MontiArc grammar in [HRR12].

6.3. Streams, I/O Relations, and Stream Processing Functions

We use the FOCUS framework [BS01] as a semantic domain of the interactive systems described using MontiArcAutomaton. FOCUS is a mathematical framework for the specification of interactive distributed systems. A central idea of the FOCUS approach is to describe component behavior as observable interactions on channels between systems and subsystems. Systems and subsystems are formalized as components with defined input and output interfaces. The behavior of a component is defined as the possible output histories produced for a given input history. These histories are formally described as streams [BS01].

In this section we introduce message streams, I/O relations for defining the behavior of components, stream processing functions as an alternative formalization, component composition, and refinement.

6.3.1. Streams of Messages

The behavior of a component can be described as its observable reaction to input by corresponding output. The complete histories of messages sent on input and output ports of a component are formalized as message streams. For each port, a message stream captures the messages in the order they occur [BS01]. A stream may be a finite or an infinite sequence. The type of finite streams of messages over the alphabet M is denoted as M^* whereas $M^\omega = M^* \cup M^\infty$ denotes finite as well as infinite streams of messages over the alphabet M.

Examples of streams over the alphabet `Boolean` = $\{\mathbf{true}, \mathbf{false}\}$ are $\langle \mathbf{true}, \mathbf{true}, \mathbf{false}, \mathbf{true} \rangle \in \mathtt{Boolean}^* \subseteq \mathtt{Boolean}^\omega$ and the infinite stream $\mathbf{true}^\infty \in \mathtt{Boolean}^\omega$. Many different kinds of streams are known from FOCUS and other frameworks with stream-based semantics (see [MS96, Bro97a, Ste97, RR11] for more examples). We employ a discrete model of streams where each element on the stream is indexed by a number $n \in \mathbb{N}$. This model of discrete message streams allows a simple abstraction of time known as time-synchronous streams. The time-synchronous model of streams models pulsed systems with a global clock and synchronous time behavior of all subsystems. The messages sent (the event observed) at a discrete time $n \in \mathbb{N}$ are recorded at position n of the message streams formalizing the communication history of the system.

The time-synchronous model of FOCUS requires all streams to be infinite since time never stops. For the purpose of behavior specifications it is still possible to use finite streams as prefixes of infinite time-synchronous streams. There are multiple possibilities

Operations on streams $s, s' \in M^\omega$, message $m \in M$, and $n \in \mathbb{N}_\infty$:

Notation	Signature	Functionality
$\langle\rangle$	$empty : M^\omega$	empty stream
$m{:}s$	$cons : M \times M^\omega \to M^\omega$	append first element
$s{\frown}s'$	$conc : M^\omega \times M^\omega \to M^\omega$	concatenation of streams
$s \sqsubseteq s'$	$pref : M^\omega \times M^\omega \to \mathbb{B}$	prefix relation
$\#s \in \mathbb{N}_\infty$	$len : M^\omega \to \mathbb{N}_\infty$	length of stream
$s.n$	$nth : \mathbb{N} \times M^\omega \to M$	n^{th} element of stream
	$hd : M^\omega \to M$	first element of stream: $s.0$
	$rt : M^\omega \to M^\omega$	stream without first element
$s\|_n$	$take : \mathbb{N}_\infty \times M^\omega \to M^\omega$	prefix of length n
m^n	$ntimesm : \mathbb{N}_\infty \times M \to M^\omega$	message iterated n times
s^n	$ntimes : \mathbb{N}_\infty \times M^\omega \to M^\omega$	stream iterated n times

Table 6.10.: Some operations on discrete message streams from [RR11].

to encode time information in message streams. One possibility is the addition of a special symbol $\surd$ (called tick) to denote the end of a time slice (time between two ticks of the global clock). In the time-synchronous streams of our semantics definition of MontiArcAutomaton models every time slice contains at most one message. We thus omit ticks between all time slices and use the special message ε to denote the absence of a message in a time slice.

The modeling language MontiArcAutomaton is not limited to time-synchronous streams. Other types of message streams are discrete timed streams of FOCUS of the form $\mathbb{N} \to M^*$ that allow to model timed systems where multiple messages may be received and sent in one time slice or dense streams $\mathbb{R}_+ \to M$ [MS96, Bro97b] and super dense streams $\mathbb{R}_+ \to M^*$ [MMP91, LML06] where time is not required to be discrete. An event-based application of MontiArcAutomaton for modeling and code generation based on dense timed streams is described in [Mar12]. Depending on the problem domain and run-time system used, one may chose the most appropriate kind of streams. We have chosen time-synchronous streams which allow automated verification (see Chapter 7) and have proven useful in a case study on model-based robotics development (see [RRW13a] and Chapter 8).

Some operations on streams are shown in Table 6.10. The empty stream is denoted as $\langle\rangle$. Appending an element $m \in M$ as the first element to a stream $s \in M^\omega$ is denoted $m{:}s$. We use the prefix order $s \sqsubseteq s'$ for streams $s \in M^*$ and $s' \in M^\omega$ when s is a prefix of s', i.e., $\exists r \in M^\omega : s{\frown}r = s'$. For $n \in \mathbb{N}$ the n^{th} element of a stream s is denoted $s.n$. The first element is $s.0$. The operation $take : \mathbb{N}_\infty \times M^\omega \to M^\omega$ applied to a stream $s \in M^\omega$ with $\#s \geq n$ yields the prefix $s|_n \sqsubseteq s$ with $\#(s|_n) = n$ or $s|_n = s$ for $\#s < n$. The *take* operation is useful for specifications and restrictions of infinite to finite message streams.

6.3.2. Component Behavior Specifications as I/O Relations and Stream Processing Functions

The behavior of a component in FOCUS is specified as its observable I/O behavior. As a concrete example for an elementary specification in FOCUS, consider the component `TogglSwitch` shown in Figure 6.11.

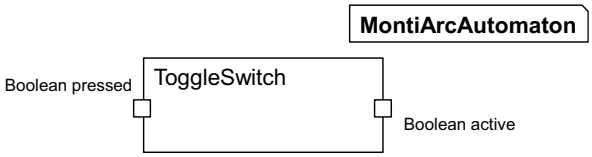


Figure 6.11.: The component `ToggleSwitch` with input port `pressed` of type `Boolean` and output port `active` of type `Boolean`.

The component has an input port `pressed` and an output port `active` both of type `Boolean`. The component reads a stream of `Boolean` values on the port `pressed`. Initially the value on port `active` is set to **`false`** meaning that the toggle switch is turned off. Once the value **`true`** is received on port `pressed` the component switches the output to **`true`**. If the component receives the value **`false`** the switch is still turned on and the component continues sending the value **`true`**. It toggles the output value on port `active` any time that it receives the input **`true`**, i.e., that a button is pressed.

An observation of the component's behavior is described by the following streams, which are valid infinite observations of the component's I/O behavior. Please note that the component `ToggleSwitch` produces the initial output **`false`** that delays its reaction to inputs by one time slice:

$$pressed = \langle \mathbf{false}, \mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false}, \mathbf{true}, \rangle \frown \mathbf{false}^{\infty}$$
$$active = \langle \mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{true}, \mathbf{true}, \mathbf{true}, \mathbf{false} \rangle \frown \mathbf{false}^{\infty}$$

A formal description of the component's I/O behavior is given in the FOCUS specification frame shown in Figure 6.12. The specification starts with declaring the component's inputs and outputs as infinite message streams of the types of the ports. We use infinite streams since we describe the behavior of a time-synchronous reactive system. The body of the specification is a predicate over the input and output streams declared in the head of the specification.

The semantics of the specification in Figure 6.12 is defined by a schematic translation into the predicate $\Phi(i, o) = i \in \texttt{Boolean}^{\infty} \wedge o \in \texttt{Boolean}^{\infty} \wedge (active.0 = \mathbf{false} \wedge \forall n \in \mathbb{N} : pressed.n = \mathbf{true} \Leftrightarrow active.(n+1) = \neg active.n)$ (see [BS01, Section 5.2.3]) that characterizes a relation $R \subseteq \texttt{Boolean}^{\infty} \times \texttt{Boolean}^{\infty}$. The relation R is referred to as the I/O behavior of the specification.

For multiple inputs and outputs of the specification we denote the I/O behavior as $R \subseteq \vec{I}^{\infty} \times \vec{O}^{\infty}$, where $\vec{I}$ is the product of all input types and $\vec{O}$ is the product of all output types.

in Boolean *pressed* **out** Boolean *active*
$active.0 = \texttt{false} \wedge$ $\forall n \in \mathbb{N} : pressed.n = \texttt{true} \Leftrightarrow active.(n+1) = \neg active.n$

Figure 6.12.: A FOCUS specification for the I/O behavior of the component ToggleSwitch from Figure 6.11. Following the notation used in [BS01] the names of inputs are given with their types in the head of the specification and the names are used as streams of these types in the body of the specification.

A number of specification mechanism for component behavior has been developed for the FOCUS framework. One example is the direct definition of the relation between inputs and outputs as shown in the specification in Figure 6.12. Other examples are assumption and guarantee style predicates, description of possible behaviors as sets of functions, state-based description of behavior using automata, and composed specifications that combine multiple specifications [Rum96, BS01, RR11].

Not all specified I/O behaviors are realizable in the sense that they are computable and can be implemented in a component. In the FOCUS framework a specification with the I/O behavior $R \subseteq \vec{I}^{\infty} \times \vec{O}^{\infty}$ is realizable if and only if at least one causal winning strategy $\tau : \vec{I}^{\infty} \to \vec{O}^{\infty}$ exists. A function τ is a winning strategy if and only if $\forall \vec{i} \in \vec{I} : (\vec{i}, \tau(\vec{i})) \in R$. A winning strategy is called causal if it is weakly causal, meaning

$$\forall \vec{i}, \vec{i}' \in \vec{I}^{\infty} \;\; \forall n \in \mathbb{N} : \vec{i}|_n = \vec{i}'|_n \Rightarrow \tau(\vec{i})|_n = \tau(\vec{i}')|_n,$$

or if the winning strategy is strongly causal, meaning

$$\forall \vec{i}, \vec{i}' \in \vec{I}^{\infty} \;\; \forall n \in \mathbb{N} : \vec{i}|_n = \vec{i}'|_n \Rightarrow \tau(\vec{i})|_{n+1} = \tau(\vec{i}')|_{n+1}$$

Weak causality in the time-synchronous model means that the output of a component up to time $n \in \mathbb{N}$ is determined by its input received until time n. Thus, the output of the component may not depend on any input the component receives in the future after time n. Strong causality implies weak causality and further strengthens the requirement: the output until time $n+1$ may only depend on the input received until time n. This models computation time: a component requires at least one time cycle for the computation triggered by an input before it can respond with an output.

The behavior of the component ToggleSwitch is deterministic and total, i.e., for every possible input stream $i \in \texttt{Boolean}^{\infty}$ there is exactly one defined output stream. We can thus specify the behavior of the component as a total function from the stream of messages on its input port to the stream of messages on its output port. One possible description is the recursive definition of a function $ts : \texttt{Boolean}^{\infty} \to \texttt{Boolean}^{\infty}$ using the helper function $tsh : \texttt{Boolean} \to \texttt{Boolean}^{\infty} \to \texttt{Boolean}^{\infty}$ to keep track of the

toggle state using the first parameter $t \in \texttt{Boolean}$:

$$\begin{aligned} ts(bs) &= \texttt{false} : tsh(\texttt{false}, bs) \\ tsh(t, \texttt{true} : bs) &= (\neg t) : tsh(\neg t, bs) \\ tsh(t, \texttt{false} : bs) &= t : tsh(t, bs) \end{aligned}$$

The specification above is complete and can thus also be seen as an implementation of the component `ToggleSwitch`. The function ts implementing component `ToggleSwitch` is strongly causal since its output stream up to position $n+1$ is completely determined by a prefix of the input stream of length n.

Not all functions describe valid implementations of components. For example the function $chgPast : \texttt{Boolean}^\infty \to \texttt{Boolean}^\infty$ with the following partial behavior definition does not describe realizable component behavior:

$$chgPast(x) = \begin{cases} \texttt{false}^\infty & \text{if } \exists t \in \mathbb{N} : x.t = \texttt{false} \\ \texttt{true}^\infty & \text{otherwise} \end{cases}$$

The function $chgPast$ is not weakly causal. For the stream $i = \texttt{true}^\infty$, the natural number $0 \neq n \in \mathbb{N}$, and the stream $i' = \texttt{true}^n \frown \texttt{false} \frown \texttt{true}^\infty$ the function $chgPast$ contradicts weak causality: $i|_n = i'|_n \wedge chgPast(i)|_n \neq chgPast(i')|_n$. Intuitively, a time-synchronous component realizing the function $chgPast$ would need to produce an output in the first time cycle without knowing whether it will receive the value **false** at a later time cycle. Either way the component would be required to change the past if the value **false** appears later on or if it never appears.

We will call the strategies $\tau : \vec{I}^\infty \to \vec{O}^\infty$ time-synchronous Stream Processing Functions (SPF) [Rum96, RR11]. By definition stream processing functions are realizable functions, i.e., timed SPF are required to be weakly causal. Untimed SPF also allow finite input and finite output streams. For the untimed case weak causality is replaced by the concept of continuity defined for functions on the complete partial orders $(\vec{I}^\omega, \sqsubseteq)$ and $(\vec{O}^\omega, \sqsubseteq)$ with the prefix order $\sqsubseteq$ on streams from Table 6.10 extended point-wise for all input and output streams in $\vec{I}$ and $\vec{O}$ [BDD+92, Rum96].

Instead of specifying the I/O behavior of a component as a relation $R \subseteq \vec{I}^\infty \times \vec{O}^\infty$ we can also specify the behavior of the component as an SPF as seen in the example for the component `ToggleSwitch`. In case a specification allows alternative behaviors it is given as a set of SPF. Each SPF describes one possible behavior implementation of the component [RR11].

6.3.3. Component Composition

Composition is one of the most important concepts for systems engineering since it allows the tackling of larger and more complex problems by the composition of solutions for smaller well-understood problems (see, e.g., [BR07] or [TMD09]). The composition of

components and the connectors between them is part of the architectural configuration of the system to accomplish the system's objectives [TMD09].

One of the key features of FOCUS is the composition of specifications to composed specifications. This allows the independent development, generalization, and reuse of component specifications for the specification of complex subsystems and systems. Composite specifications in FOCUS may be defined in graphical style, constraint style, and operator style. Out of these three kinds we focus on the graphical style that we use for component composition in the modeling languages MontiArc and MontiArcAutomaton and the constraint style that is close to our implementation for the composition of specifications in the MontiArcAutomaton analysis framework presented in Chapter 7.

An example of the graphical style is given in Figure 6.13 where the specification of the component `BumperBotESController` is graphically represented as the composition of the specifications of the components `BumpControlES` and `Timer`.

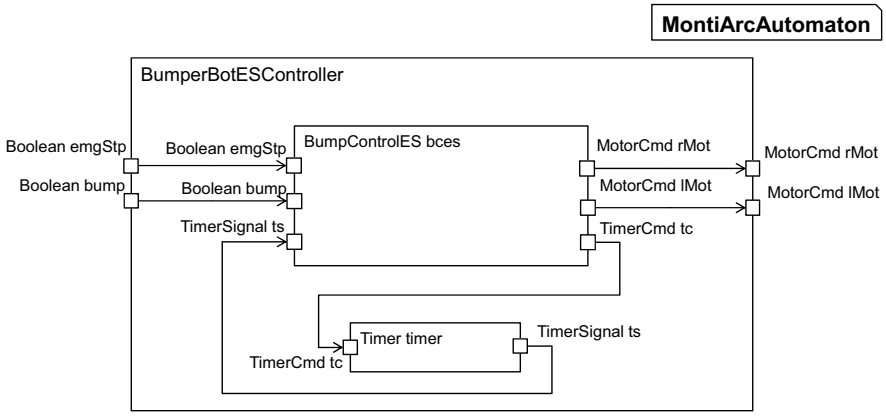


Figure 6.13.: The composed component `BumperBotESController` with its subcomponents `BumpControlES` and `Timer`.

The same example of the composed specification `BumperBotESController` can be formulated in the constraint style as shown in Figure 6.14. The head of the FOCUS composed specification contains input channels (one for each input port of the composed component), output channels (one for each output port), and local channels (one for each connector between subcomponents). The body of the specification in Figure 6.14 consists of constraints that constrain the input, local, and output channels of the specification based on the composition and the specifications of the subcomponents. Here the specifications of the subcomponents are interpreted as assignments of input histories to output histories [BS01].

The semantics of composite specifications, which is also the semantics of component composition in MontiArc and MontiArcAutomaton, is the existential quantification of streams for the local channels and the conjunction of all subspecifications. In the example of the specification of the component `BumperBotESController`, the denotational

in Boolean *emgStp*, Boolean *bump*
out MotorCmd *rMot*, MotorCmd *lMot*
loc TimerCmd *tc*, TimerSignal *ts*
$(rMot, lMot, tc) :=$ BumpControlES$(emgStp, bump, ts)$
$(ts) :=$ Timer(tc)

Figure 6.14.: A FOCUS composite specification of the component BumpControlES from Figure 6.13 given in constraint style.

semantics of the composition in Figure 6.13 and Figure 6.14 is given as the following predicate:

$$\Phi_{\texttt{BumperBotESController}}(emgStp, bump, rMot, lMot) = \\ \exists tc, ts : \Phi_{\texttt{BumpControlES}}(emgStp, bump, ts, rMot, lMot, tc) \wedge \Phi_{\texttt{Timer}}(tc, ts)$$

In the MontiArcAutomaton modeling language component composition is defined in component type definitions from Definition 6.8. A component definition does not explicitly list its local channels, as necessary in FOCUS composite specifications. The analogue concept to local channels in FOCUS composite specifications are the connectors in component definitions. It is not necessary to introduce an existentially quantified message stream for every connector in a composed component definition. Multiple connectors with the same source (subcomponent and port) require only a single source stream.

The specification of a composed component is composed of the specifications of its subcomponents. In the I/O relation semantics of a composed component type definition all specifications are given as relations $R \subseteq \vec{I}^{\infty} \times \vec{O}^{\infty}$ where $\vec{I}^{\infty}$ is the input on all input ports of the component with $\vec{I} = \times_{p \in CPorts_{IN}} p.type$ and $\vec{O}^{\infty}$ is the output on all output ports of the component with $\vec{O} = \times_{p \in CPorts_{OUT}} p.type$. We formally define the I/O relation semantics of MontiArcAutomaton composed component types from Definition 6.8 based on the I/O relation semantics of their subcomponents in Definition 6.15. The semantics definition starts with the existential quantification of the streams on local channels (variables $l_{(sub,p)}$ in Definition 6.15) — here, all output ports of all subcomponents. Finally, a pair of inputs and outputs is in the semantics of the component if and only if the inputs and outputs selected according to subcomponent composition are in the I/O relation semantics of the subcomponents.

The output out_{sub} of the subcomponent sub constrains the local channel streams $l_{(sub,p)}$ (see Definition 6.15, Item 1).

The input streams $(in_{sub})_p$ of the subcomponents sub and their input ports p are determined according to three cases in Definition 6.15, Item 2. For all subcomponents $sub \in CSubCmps$ and their input ports $p \in sub.type.CPorts_{IN}$ we distinguish: (a) the stream is provided via a connector from an incoming port of the parent component, (b) the input is provided from a local channel $l_{(sc,sp)}$ identified by a source component sc and the source port sp, or (c) the input port is not connected and thus set to an infinite stream of empty messages.

Finally, for the streams on the output ports $p \in CPorts_{OUT}$ of the parent component we distinguish two cases (see Definition 6.15, Item 3): (a) the output stream is assigned the value of the local channel variable of the connected port of a subcomponent or (b) the output stream is the same as one of the input streams.

Definition 6.15 (I/O relation semantics for composed components)**.** The I/O relation semantics of a composed component type with the subcomponent and output port pairs $(sub_1, p_1), .., (sub_n, p_n) \in \{(sub, p) \mid sub \in CSubCmps, p \in sub.type.CPorts_{OUT}\}$ is the relation $R \subseteq \vec{I}^{\infty} \times \vec{O}^{\infty}$ with $\vec{I} = \bigtimes_{p \in CPorts_{IN}} p.type$ and $\vec{O} = \bigtimes_{p \in CPorts_{OUT}} p.type$, where

$$R(in, out) \Leftrightarrow \exists l_{(sub_1,p_1)} \in p_1.type^{\infty}, .., l_{(sub_n,p_n)} \in p_n.type^{\infty} : \quad \text{(i) } \textit{local channels}$$
$$\forall sub \in CSubCmps : R_{sub.type}(in_{sub}, out_{sub}) \quad \text{(i) } \textit{subcomponents}$$

with the variables in_{sub} and out_{sub} assigned according to subcomponent composition

1. $\forall p \in sub.type.CPorts_{OUT} : (out_{sub})_p = l_{(sub,p)}$ (i) *output to local channel*
2. $\forall p \in sub.type.CPorts_{IN} : (in_{sub})_p \in p.type^{\infty} \wedge$
$$(in_{sub})_p = \begin{cases} in_{pp} \text{ if } \exists con \in CCons : con.tgtCmp = sub.name \wedge \\ \qquad con.tgtPort = p \wedge con.srcCmp = cType \wedge \\ \qquad con.srcPort = pp \quad \text{(i) } \textit{parent-to-child} \\ l_{(sc,sp)} \text{ if } \exists con \in CCons : con.tgtCmp = sub.name \wedge \\ \qquad con.tgtPort = p \wedge con.srcCmp = sc.name \wedge \\ \qquad con.srcPort = sp \quad \text{(i) } \textit{child-to-child} \\ \varepsilon_{\cup}^{\infty} \text{ otherwise} \quad \text{(i) } \textit{not connected} \end{cases}$$
3. $\forall p \in CPorts_{OUT} :$
$$out_p = \begin{cases} l_{(sc,sp)} \text{ if } \exists con \in CCons : con.tgtCmp = cType \wedge \\ \qquad con.tgtPort = p \wedge con.srcCmp = sc.name \wedge \\ \qquad con.srcPort = sp \quad \text{(i) } \textit{child-to-parent} \\ in_{pp} \text{ if } \exists con \in CCons : con.tgtCmp = cType = con.srcCmp \wedge \\ \qquad con.tgtPort = p \wedge con.srcPort = pp \quad \text{(i) } \textit{parent-to-parent} \end{cases}$$

△

Composition with strong and weak causality

In Section 6.3.2 we have introduced the concept of strong and weak causality for strategies τ realizing component behavior. Intuitively, a weakly causal component may react to an input instantaneously in the same global time slice. A strongly causal component may react to input received at time $t \in \mathbb{N}$ no earlier than at time $t + 1$. We repeat the definition of strong causality for a strategy $\tau : \vec{I}^{\omega} \to \vec{O}^{\omega}$ realizing component behavior:

$$\forall \vec{i}, \vec{i}' \in \vec{I}^{\infty} \ \forall n \in \mathbb{N} : \vec{i}|_n = \vec{i}'|_n \Rightarrow \tau(\vec{i})|_{n+1} = \tau(\vec{i}')|_{n+1}$$

For the composition of components it is important to distinguish between strong and weak causality. The composition of strongly causal components preserves realizability,

whereas the composition of weakly causal components may lead to unrealizable behavior. Consider the component SumUp shown in Figure 6.16. The subcomponent Add is specified by the relation R_{Add} as

$$\forall i \in \vec{I^{\infty}}, o \in \vec{O}^{\infty} : R_{Add}(i,o) \Leftrightarrow \forall t \in \mathbb{N} : i_{s1}.t + i_{s2}.t = o_{sum}.t$$

This specification has a single (weakly causal) strategy *add*:

$$\begin{array}{rcl} add : (int^{\infty} \times int^{\infty}) & \rightarrow & int^{\infty} \text{ where} \\ add(a{:}as, b{:}bs) & \mapsto & (a+b) \frown add(as, bs) \end{array}$$

Please note that the strategy *add* is not strongly causal.

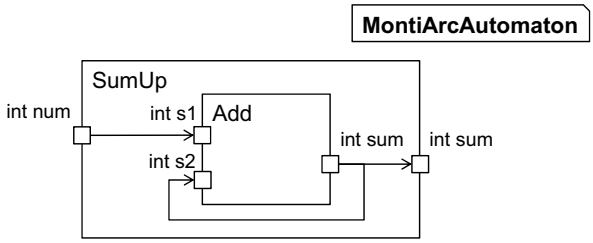


Figure 6.16.: The composed component SumUp consisting of the component Add with a feedback look.

If we compute the specification of the composed component type SumUp according to Definition 6.15 we receive the specification of the component SumUp as the relation R_{SumUp}:

$$\begin{array}{rcl} R_{SumUp}(in, out) & \Leftrightarrow & \exists l_{(add,sum)} \in int^{\infty} : \\ & & \quad R_{Add}(in_{add}, out_{add}) \wedge out_{sum} = l_{(add,sum)} \text{ where} \\ (in_{add})_{s1} & = & in_{num} \wedge \\ (in_{add})_{s2} & = & l_{(add,sum)} \wedge \\ (out_{add})_{sum} & = & l_{(add,sum)} \end{array}$$

Together with the definition of the relation R_{Add} we receive

$$\begin{array}{rcl} R_{SumUp}(in, out) & \Leftrightarrow & \exists l_{(add,sum)} \in int^{\infty} : \\ & & \quad \forall t \in \mathbb{N} : in_{num}.t + l_{(add,sum)}.t = l_{(add,sum)}.t \wedge \\ & & \quad out_{sum} = l_{(add,sum)} \end{array}$$

The constraint in line two $in_{num}.t + l_{(add,sum)}.t = l_{(add,sum)}.t$ is created due to the immediate feedback of the weakly causal component Add. It can only be satisfied for

$in_{num}.t = 0$, i.e., the relation R_{SumUp} contains the single element $(0^\infty, 0^\infty)$ and is not total. Thus, there is no winning strategy that implements the behavior of the component `SumUp`. The composition shown in Figure 6.16 leads to unrealizable behavior.

In case of strong causality the composition of realizable specifications preserves realizability [BS01, RR11]. Component composition in the case of weak causality does not necessarily preserve realizability. Feedback cycles of arbitrary length, e.g., direct feedback as in the example in Figure 6.16, may lead to an unrealizable behavior specification. One solution to compose components with feedback cycles is to make sure that every feedback cycle contains at least one strongly causal component, e.g., a component that delays its output by one time cycle. For a discussion of the causality problem and various solutions in synchronous communication see also [CRT07].

6.3.4. Refinement

Refinement is the transition from an abstract specification to a more concrete specification or implementation. One of the main concepts of FOCUS is stepwise behavioral refinement. An abstract specification $S_{abstract}$ is refined by a specification $S_{concrete}$ if and only if all I/O behaviors allowed by $S_{concrete}$ are also allowed by $S_{abstract}$. Behavioral refinement may be written as a logical implication between the predicates of the specifications $\Phi_{concrete} \Rightarrow \Phi_{abstract}$ or as an implication for the I/O relations [BS01]:

$$\forall i \in \vec{I}^\infty, o \in \vec{O}^\infty : (i, o) \in R_{concrete} \Rightarrow (i, o) \in R_{abstract}$$

We denote the refinement of the specification $S_{abstract}$ to the specification $S_{concrete}$ as $S_{abstract} \rightsquigarrow S_{concrete}$. Behavioral refinement is reflexive, i.e., for all specifications S: $S \rightsquigarrow S$, and transitive, i.e., for all specifications S_1, S_2, and S_3: $S_1 \rightsquigarrow S_2 \wedge S_2 \rightsquigarrow S_3 \Rightarrow S_1 \rightsquigarrow S_3$.

The refinement of specifications based on SPF is defined as the inclusion of the set of functions of the more concrete specification in the set of functions in the abstract specification [RR11]. Given the specification $F_{abstract}$ as a set of SPF and the specification $F_{concrete}$ as a set of SPF, the specification $F_{concrete}$ refines $F_{abstract}$ if and only if

$$F_{concrete} \subseteq F_{abstract}$$

As before, this type of refinement based on set inclusion for sets of SPF is reflexive and transitive.

In FOCUS refinement is compatible with the composition of specifications [Bro93, BS01]. A refinement of one of the specifications leads to a refinement of the composition of the specifications. Thus, we have a suitable theory to, e.g., specify the behavior of a system as a MontiArcAutomaton automaton and freely decompose the implementation while still being able to reason about the refinement of the specification by implementations.

In some cases an implementation may refine not only the behavior of a specification but also the interface of the component or subsystem under consideration. One example would be adding an additional output in the implementation for monitoring. Another example might be extending a subsystem with additional inputs and outputs to fulfill

related or independent tasks. In these cases a specification of the behavior of a system might still be valid but the existing definition of refinement does not apply anymore. Broy [Bro93] has extended the refinement notion to various cases of interface refinements to address this problem.

We are particularly interested in the interface refinement case of upward simulation. This case is illustrated in Figure 6.17 for two components `abstract` and `concrete`. The component `abstract` has as input an abstraction of the concrete input that the component `concrete` receives. The input abstraction is depicted as A. To be in an upward simulation refinement relation, the output of the component `concrete`, after applying the output abstraction $\tilde{A}$, matches the output of the component `abstract`.

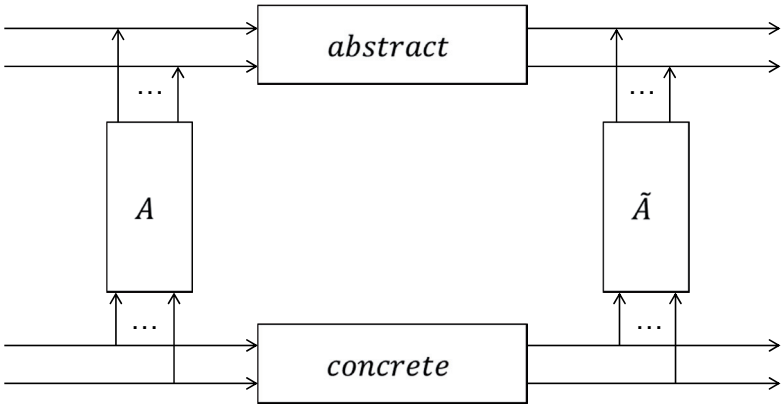


Figure 6.17.: Upward simulation for behavior refinement of components in combination with interface refinement (see [Bro93]).

The upward input and output abstractions A and $\tilde{A}$ may be any kind of FOCUS specification [Bro93]. In the two scenarios above and in the remainder of this thesis we limit ourselves to projections over the set of input streams that may only remove streams from the input and the output respectively. An example of this kind of abstractions is shown in Figure 6.18. The abstractions A and $\tilde{A}$ are implemented as composed components that implement the projection by only forwarding some of their input streams.

It is easy to see that the two abstractions A and $\tilde{A}$ shown in Figure 6.18 composed with the specification `BumpControlSpec1a` and the component `BumpControl` result in specifications with the same interface (see top and bottom of Figure 6.18). We can thus apply the general notion of behavior refinement to the compositions of the components with the two abstractions A and $\tilde{A}$.

For the purpose of this thesis we only consider abstractions that remove inputs and outputs and thus can be easily derived from the matching names of the ports of the components under consideration. Intuitively, with our special form of the upward simulation refinement, the specification ignores additional inputs and does not constrain additional outputs. We consider this restricted kind of upward simulation useful when only a sub-

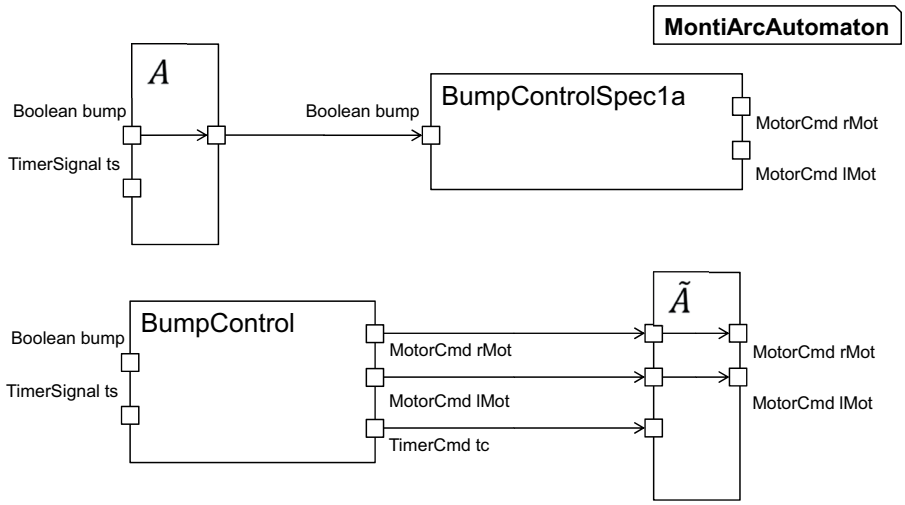


Figure 6.18.: A concrete example of two abstractions A and $\tilde{A}$ for upward simulation refinement of the specification `BumpControlSpec1a` by the component `BumpControl` (see the example in Section 7.1).

set of all ports of a component are relevant for a specification or if the interface of a component expands during evolution but existing specifications should still be checked for refinement. Some examples are available in Section 7.1.

6.4. MAA$_{ts}$: a MontiArcAutomaton Language Profile for Time-Synchronous Communication

In the previous sections we have presented the syntax of the modeling language MontiArcAutomaton and the framework FOCUS which serves as a semantic domain for describing component behavior and interaction. We now define a subset of the modeling language MontiArcAutomaton, on which we focus for the rest of this thesis. The subset is a language profile [CGR09] of MontiArcAutomaton for time-synchronous communication (short MAA$_{ts}$). It restricts the automata inside components to specify and implement strongly causal behavior based on discrete message streams. The profile only restricts the structure of the automata inside components but not the language elements for component definition and composition.

We formally define the structure of MontiArcAutomaton automata in the MAA$_{ts}$ profile in Definition 6.19.

Definition 6.19 (MAA_{ts} automaton)**.** A MAA_{ts} automaton for a component type definition $cmp \in CTDefs$ (see Definition 6.8) is a tuple $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ where

- S is a non-empty finite set of states,
- $\vec{I} = \times_{p \in cmp.CPorts_{IN}} \vec{I}_p$, where $\vec{I}_p = p.type$ is the type of input port p,
- $\vec{O} = \times_{p \in cmp.CPorts_{OUT}} \vec{O}_p$, where $\vec{O}_p = p.type$ is the type of output port p,
- $\vec{V} = \times_{var \in cmp.CVars} \vec{V}_{var}$, where $\vec{V}_{var}$ is the type of local variable var,
- $\vec{\gamma} \in \vec{V}$ is the initial assignment to local variables,
- δ is a transition relation, and
- $\iota \subseteq (S \times \vec{O})$, where $(s, \vec{o}) \in \iota$ is an initial state $s \in S$ and its initial output.

We require $\vec{\varepsilon}_{\cup} \in \vec{I}$ and $\vec{\varepsilon}_{\cup} \in \vec{O}$, i.e., all input and output alphabets contain the special semantic value $\varepsilon_{\cup}$ that denotes the absence of a message.
The transition relation δ is a set of tuples $(s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a})$, where

- $s_{src} \in S$ is a source state,
- $\vec{i} \in \vec{I}$ and for $p \in cmp.CPorts_{IN}$ the value on input port p is $\vec{i}_p \in \vec{I}_p$,
- $\vec{v} \in \vec{V}$ and for $var \in cmps.CVars$ the value of local variable var is $\vec{v}_{var} \in \vec{V}_{var}$,
- $s_{tgt} \in S$ is a target state,
- $\vec{o} \in \vec{O}$ and for $p \in cmp.CPorts_{OUT}$ the value on output port p is $\vec{o}_p \in \vec{O}_p$, and
- $\vec{a} \in \vec{V}$ and for $var \in cmp.CVars$ the assignment to local variable var is $\vec{a}_{var} \in \vec{V}_{var}$.

△

The language profile presented in Definition 6.19 restricts the modeling language MontiArcAutomaton presented in Section 6.2 to one message on every input and output port. This allows a direct definition of a strongly causal time-synchronous semantics. The inputs on a transition are the inputs read in the current time cycle and the outputs defined by the transition are the outputs sent in the next time cycle.

Modeling MAA_{ts} automata with MontiArcAutomaton according to the format shown in Definition 6.19 would require the modeler to always specify input and output messages on all ports even if only some of the messages are required to enable a transition. Also, the structure given in Definition 6.19 does not contain guard predicates to conveniently specify multiple valid input values and variable values combinations. We thus extend the basic structure of MAA_{ts} automata with syntactic features to $^*\text{MAA}_{ts}$ automata in Definition 6.20.

Definition 6.20 ($^*\text{MAA}_{ts}$ automata)**.** $^*\text{MAA}_{ts}$ automata extend MAA_{ts} automata with references to ports, variables, and the symbol $*$ to denote syntactically unspecified values. We denote the extension of all alphabets $\vec{A}_k$ in $\vec{A}$ with the symbol $*$ as $^*\vec{A} = \times_k(\vec{A}_k \cup \{*\})$. The difference between the structure of MAA_{ts} automata and $^*\text{MAA}_{ts}$ automata is:

- $\vec{\gamma} \in {}^*\vec{V}$ and for $var \in cmp.CVars$ the (possibly unspecified) initial value of variable var is $\vec{\gamma}_{var}$,

- $\iota \subseteq (S \times {}^*\vec{O})$, where for $(s, \vec{o}) \in \iota$ and for $p \in cmp.CPorts_{OUT}$ the (possibly unspecified) initial output on port p is $\vec{o}_p$.

The transition relation δ is extended with guard predicates ϕ to contain the tuples $(s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a})$. The differences to MAA_{ts} automata are:

- $\phi : \vec{I} \times \vec{V} \to \mathbb{B}$ is a guard predicate over inputs and local variables,
- $\vec{i} \in {}^*\vec{I} \cup cmp.CVars$ allows unspecified values and references to variables on input ports,
- $\vec{v} \in {}^*\vec{V} \cup cmp.CVars$ with $\forall var \in CVars : \vec{v}_{var} \neq var$ allows unspecified values and references to other variables for local variables,
- $\vec{o} \in {}^*\vec{O} \cup cmp.CVars \cup cmp.CPorts_{IN}$ allows unspecified values, references to variables, and references to input ports on output ports, and
- $\vec{a} \in {}^*\vec{V} \cup cmp.CVars \cup cmp.CPorts_{IN}$ allows unspecified assignments, references to variables, and references to input ports for local variables.

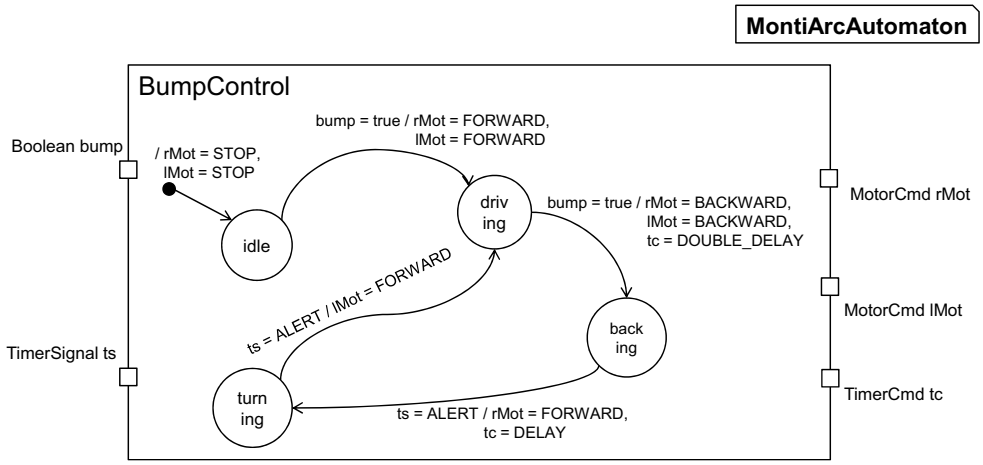
△

Definition 6.20 does not require every transition specified by the transition relation δ to provide an assignment of values on all ports and local variables (denoted by the symbol $*$). This incomplete information in ${}^*\text{MAA}_{ts}$ automata is interpreted as syntactic underspecification of the modeled component. Handling syntactic underspecification is part of the semantics definition of the MAA_{ts} language profile.

Please note the difference between the ε message (introduced in Definition 6.19 to be an element of every input and output alphabet) and the missing value denoted by $*$. While the former is an element contained in the input and output alphabets, the latter is only used on the structure definition level to mark underspecification. The symbol $*$ only exists in ${}^*\text{MAA}_{ts}$ automata, while ε is part of the semantics of MAA_{ts} automata to denote the absence of a message.

Figure 6.21 repeats the ${}^*\text{MAA}_{ts}$ model of component `BumpControl` from Section 6.1 and shows the transition relation δ of the automaton inside the component following Definition 6.19. This illustrates the connection between MontiArcAutomaton models in concrete syntax and their structural representation including the symbol $*$.

In addition to adding the symbol $*$ to ${}^*\text{MAA}_{ts}$ automata we also allow the inputs $\vec{i}$, variable values $\vec{v}$, outputs $\vec{o}$, and variable assignments $\vec{a}$ of the automaton's transitions to refer to variable names or input and output ports respectively. This is a convenience mechanism for modelers to model transitions where, e.g., the input read on one port should correspond to the value of a local variable. As another example, consider forwarding the value read on an input port via an output port of the component (the output $\vec{o}_p$ then equals the name of the corresponding input port on a transition of the ${}^*\text{MAA}_{ts}$ automaton). In Definition 6.20 we assume that names of variables and ports can be distinguished from values of the corresponding variables or ports that reference the names on a transition. In the concrete syntax of the MontiArcAutomaton implementation this is ensured by context conditions [RRW14].



s_{src}	$\vec{i}_{bump}$	$\vec{i}_{ts}$	s_{tgt}	$\vec{o}_{rMot}$	$\vec{o}_{lMot}$	$\vec{o}_{tc}$
idle	**true**	*	driving	FORWARD	FORWARD	*
driving	**true**	*	backing	BACKWARD	BACKWARD	DOUBLE_DELAY
backing	*	ALERT	turning	FORWARD	*	DELAY
turning	*	ALERT	driving	*	FORWARD	*

Figure 6.21.: The MontiArcAutomaton component `BumpControl` with a *MAA$_{ts}$ automaton (upper part, also shown in Figure 6.3) and its transition relation δ according to Definition 6.20. To fit all entries in the table we omitted the guard ϕ = **true** and the empty list of variables $\vec{v}$.

Guards in *MAA$_{ts}$ automata are predicates over inputs and local variable values. A transition can only be taken if its guard predicate evaluates to **true**. Guards are a convenience element of the MontiArcAutomaton language. In MAA$_{ts}$ automata, without guard predicates ϕ in the transition relation, guards may be expressed as the (possibly infinite) set of concrete value combinations that satisfy the guard predicate.

The language elements added to MAA$_{ts}$ automata in Definition 6.20 do not increase the expressiveness of MAA$_{ts}$ automata. We show a syntactic translation of *MAA$_{ts}$ automata into MAA$_{ts}$ automata from Definition 6.19. This translation consists of three steps: (1) remove references from and to variables and ports (see Section 6.4.1), (2) enabledness expansion of the transition system (see Section 6.4.2), and (3) completion of the transition system. The completion of the transition system as a removal of syntactic underspecification via the symbol $*$ is a semantic variation point [GR10]. Depending on the usage of the model different semantics might be intended. We handle three different cases. In Section 6.4.3 we complete the transition system for using *MAA$_{ts}$ automata as implementations. In Section 6.4.4 we present two completions for using *MAA$_{ts}$ automata as specifications.

6.4.1. Removing References

To remove references to other elements from $^*\text{MAA}_{ts}$ automata transitions we replace the transitions that include references by transitions that unfold these references with concrete values. In case the value of the referenced element is given on the transition, the reference is replaced by that value. In case the transition does not specify a value for the referenced entity, we replace the transition with all possible transitions where the referenced and the referencing elements have the same value. This replacement is formalized as the function *removeReferences* in Definition 6.22.

The function definition handles the four main cases for references in inputs $\vec{i}$, variable values $\vec{v}$, outputs $\vec{o}$, and variable assignments $\vec{a}$ according to Definition 6.20. For all four cases the function considers two or three subcases for each value or reference. There are two cases for inputs $\vec{i}$ and variable values $\vec{v}$. These may be given as references to variables or as values. There are three cases for outputs $\vec{o}$ and variable assignments $\vec{a}$. These may be given as references to variables, as references to input ports, or as values.

In each of the four steps in Definition 6.22 the transition relation is defined based on the previous transition relation. For a single transition from the previous transition relation possibly multiple transitions are added to the new relation. The transitions may only differ in the elements that are introduced with a primed name in the definition of the relation in each step. All other elements are fixed. The last three steps require a possible expansion of more than one element of each transition. The last lines in the definition of the new transition relation then fix the values that should not be expanded because they are not referenced.

The reference removal is organized in four steps in Definition 6.22, Item 1-Item 4. The function *removeReferences* first removes all references from elements possibly referenced in later steps. This ensures that references evaluate to either ε or concrete values. The first step in Definition 6.22, Item 1 forms an exception. Variables $\vec{v}$ might reference variables and thus one iteration of the set comprehension could replace a reference by a reference. In this case the set comprehension needs to be iterated again with the intermediate transition relation from the first evaluation instead of the original transition relation δ. Each iteration removes references. After at most $|cmp.CVars|$ iterations of Definition 6.22, Item 1, all references from local variables to local variables are removed.

Definition 6.22 ($^*\text{MAA}_{ts}$ automata reference removal)**.** The function *removeReferences* updates the transition relation δ of $^*\text{MAA}_{ts}$ automata by removing references to variables and input ports on transitions

$$removeReferences : (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota) \mapsto (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta', \iota)$$

where

1. remove references from variables to variables (iterate at most $|cmp.CVars|$ times);
   $\delta'''' = \{(s_{src}, \phi, \vec{i}, \vec{v}', s_{tgt}, \vec{o}, \vec{a}) \mid (s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta,$
   $\forall var \in cmp.CVars :$
   if($\vec{v}_{var} \in cmp.CVars$**) then** *reference to other variable*

if $(\vec{v}_{(\vec{v}_{var})} = *)$ **then** $\vec{v}'_{var} = \vec{v}'_{(\vec{v}_{var})} \in \vec{V}_{var}$ **else** $\vec{v}'_{var} = \vec{v}_{(\vec{v}_{var})} = \vec{v}'_{(\vec{v}_{var})}$
else
$\vec{v}'_{var} = \vec{v}_{var}$
}

2. remove references from input ports to variables
$\delta''' = \{(s_{src}, \phi, \vec{i}', \vec{v}', s_{tgt}, \vec{o}, \vec{a}) \mid (s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta''''$,
$\forall p \in cmp.CPorts_{IN}$:
if $(\vec{i}_p \in cmp.CVars)$ **then** *reference to variable from input port*
if $(\vec{v}_{(\vec{i}_p)} = *)$ **then** $\vec{i}'_p = \vec{v}'_{(\vec{i}_p)} \in \vec{V}_{(\vec{i}_p)}$ **else** $\vec{i}'_p = \vec{v}_{(\vec{i}_p)} = \vec{v}'_{(\vec{i}_p)}$
else
$\vec{i}'_p = \vec{i}_p$,
preserve variable values not referenced
$\forall var \in \{var \in cmp.CVars \mid \nexists p \in cmp.CPorts_{IN} : \vec{i}_p = var\} : \vec{v}'_{var} = \vec{v}_{var}$
}

3. remove references from output ports to variables and input ports
$\delta'' = \{(s_{src}, \phi, \vec{i}', \vec{v}', s_{tgt}, \vec{o}', \vec{a}) \mid (s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta'''$,
$\forall p \in cmp.CPorts_{OUT}$:
if $(\vec{o}_p \in cmp.CVars)$ **then** *reference to variable from output port*
if $(\vec{v}_{(\vec{o}_p)} = *)$ **then** $\vec{o}'_p = \vec{v}'_{(\vec{o}_p)} \in \vec{V}_{(\vec{o}_p)}$ **else** $\vec{o}'_p = \vec{v}_{(\vec{o}_p)} = \vec{v}'_{(\vec{o}_p)}$
else if $(\vec{o}_p \in cmp.CPorts_{IN})$ **then** *reference to input from output*
if $(\vec{i}_{(\vec{o}_p)} = *)$ **then** $\vec{o}'_p = \vec{i}'_{(\vec{o}_p)} \in \vec{I}_{(\vec{o}_p)}$ **else** $\vec{o}'_p = \vec{i}_{(\vec{o}_p)} = \vec{i}'_{(\vec{o}_p)}$
else
$\vec{o}'_p = \vec{o}_p$,
preserve variable values and inputs not referenced
$\forall var \in \{var \in cmp.CVars \mid \nexists p \in cmp.CPorts_{OUT} : \vec{o}_p = var\} : \vec{v}'_{var} = \vec{v}_{var}$,
$\forall inp \in \{inp \in cmp.CPorts_{IN} \mid \nexists p \in cmp.CPorts_{OUT} : \vec{o}_p = inp\} : \vec{i}'_{inp} = \vec{i}_{inp}$
}

4. remove references from variable assignments to variables and input ports
$\delta' = \{(s_{src}, \phi, \vec{i}', \vec{v}', s_{tgt}, \vec{o}, \vec{a}') \mid (s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta''$,
$\forall var \in cmp.CVars$:
if $(\vec{a}_{var} \in cmp.CVars)$ **then** *reference to variable in assignment*
if $(\vec{v}_{(\vec{a}_{var})} = *)$ **then** $\vec{a}'_{var} = \vec{v}'_{(\vec{a}_{var})} \in \vec{V}_{var}$ **else** $\vec{a}'_{var} = \vec{v}_{(\vec{a}_{var})} = \vec{a}'_{(\vec{a}_{var})}$
else if $(\vec{a}_{var} \in cmp.CPorts_{IN})$ **then** *reference to input*
if $(\vec{i}_{(\vec{a}_{var})} = *)$ **then** $\vec{a}'_{var} = \vec{i}'_{(\vec{a}_{var})} \in \vec{V}_{var}$ **else** $\vec{a}'_{var} = \vec{i}_{(\vec{a}_{var})} = \vec{i}'_{(\vec{a}_{var})}$
else
$\vec{a}'_{var} = \vec{a}_{var}$
preserve variable values and inputs not referenced
$\forall var \in \{var \in cmp.CVars \mid \nexists asgmt \in cmp.CVars : \vec{a}_{asgmt} = var\} : \vec{v}'_{var} = \vec{v}_{var}$,
$\forall inp \in \{inp \in cmp.CPorts_{IN} \mid \nexists asgmt \in cmp.CVars : \vec{a}_{asgmt} = inp\} : \vec{i}'_{inp} = \vec{i}_{inp}$
}

△

We illustrate the effect of the function *removeReferences* from Definition 6.22 to the transition system of the automaton of the component `Buffer<MotorCmd>`. The parametrized component `Buffer<T>` is shown in Listing 6.6 and the enumeration type `MotorCmd` = {`STOP`, `FORWARD`, `BACKWARD`} is shown in Figure 6.4. We show the first transition of the automaton in the top table in Figure 6.23. The transition includes two references where the element $\vec{o}_{response} = storage$ references the value $\vec{v}_{storage}$ of the local variable `storage` of the type `MotorCmd`. Also, the element $\vec{a}_{storage} = data$ references the value $\vec{i}_{data}$ of the input port `data`. In both cases the transition does not define a value for the referenced element.

The second table in Figure 6.23 shows the expansion of the transition according to Definition 6.22, Item 3. The reference $\vec{o}_{response} = storage$ has been expanded to the set of transitions where $\vec{o}_{response} = \vec{v}_{storage}$.

The third table in Figure 6.23 shows the expansion of the transition according to Definition 6.22, Item 4. The reference $\vec{a}_{storage} = data$ has been expanded to the set of transitions where $\vec{a}_{storage} = \vec{i}_{data}$.

After the application of the function *removeReferences* from Definition 6.22 all references are removed.

6.4.2. Enabledness Expansion

Definition 6.19 has added guards to the transition relation and the symbol $*$ to mark syntactic underspecification in ${}^{*}\text{MAA}_{ts}$ automata. A transition of a ${}^{*}\text{MAA}_{ts}$ automaton is enabled if and only if it matches the current input (for all $p \in cmp.CPorts_{IN}$ the value $\vec{i}_p$ equals the current input on port p or $\vec{i}_p = *$), matches the current values of local variables (for all $var \in cmp.CVars$ the value $\vec{v}_{var}$ equals the current value of the local variable var or $\vec{v}_{var} = *$), and the guard predicate ϕ is satisfied by the current input and the current values of the local variables. The symbol $*$ is basically used for input ports and variables to state that the concrete values are not relevant for the enabledness of the transition.

Based on this interpretation of the enabledness of transitions we can replace all transitions containing the symbol $*$ as part of their input or as part of the specified values for local variables by the (possibly infinite) set of transitions matching the enabledness behavior without using the symbol $*$ in the input tuples $\vec{i}$ and variable tuples $\vec{v}$. This expansion of the transition system is called enabledness expansion and defined in Definition 6.24.

In addition, the expansion of all transitions in δ to transitions where $\vec{i}$ and $\vec{v}$ no longer contain the symbol $*$ allows us to evaluate all guard predicates on all transitions. Transitions are only relevant if $\phi(\vec{i}, \vec{v}) =$ **`true`**. Definition 6.24 removes the transitions that are never enabled from the transition relation δ.

Given the following transition of the component `Buffer<MotorCmd>` from Listing 6.6, ll. 16-17 (first transition):

s_{src}	$\vec{v}_{storage}$	$\vec{i}_{request}$	$\vec{i}_{data}$	s_{tgt}	$\vec{o}_{response}$	$\vec{a}_{storage}$
buffering	*	**true**	*	buffering	storage	data

Application of *removeReferences* from Definition 6.22, Item 3 for the case $\vec{o}_{response}$ = *storage* and $\vec{v}_{storage}$ = ∗:

s_{src}	$\vec{v}_{storage}$	$\vec{i}_{request}$	$\vec{i}_{data}$	s_{tgt}	$\vec{o}_{response}$	$\vec{a}_{storage}$
buffering	STOP	**true**	*	buffering	STOP	data
buffering	FORWARD	**true**	*	buffering	FORWARD	data
buffering	BACKWARD	**true**	*	buffering	BACKWARD	data

Application of *removeReferences* from Definition 6.22, Item 4 for the case $\vec{a}_{storage}$ = *data* and $\vec{i}_{data}$ = ∗:

s_{src}	$\vec{v}_{storage}$	$\vec{i}_{request}$	$\vec{i}_{data}$	s_{tgt}	$\vec{o}_{response}$	$\vec{a}_{storage}$
b...	STOP	**true**	STOP	b...	STOP	STOP
b...	STOP	**true**	FORWARD	b...	STOP	FORWARD
b...	STOP	**true**	BACKWARD	b...	STOP	BACKWARD
b...	FORWARD	**true**	STOP	b...	FORWARD	STOP
b...	FORWARD	**true**	FORWARD	b...	FORWARD	FORWARD
b...	FORWARD	**true**	BACKWARD	b...	FORWARD	BACKWARD
b...	BACKWARD	**true**	STOP	b...	BACKWARD	STOP
b...	BACKWARD	**true**	FORWARD	b...	BACKWARD	FORWARD
b...	BACKWARD	**true**	BACKWARD	b...	BACKWARD	BACKWARD

Figure 6.23.: Example of an application of the function *removeReferences* to the first transition of the transition system of the automaton `Buffer<MotorCmd>` with the component `Buffer<T>` shown in Listing 6.6 and the enumeration type `MotorCmd` shown in Figure 6.4.

Definition 6.24 (Enabledness expansion for $^*\text{MAA}_{ts}$ automata)**.** The transition relation expansion function *enablednessExpansion* updates the transition relation δ of $^*\text{MAA}_{ts}$ automata by expanding the enabledness elements inputs, variables and guards

$$enablednessExpansion : (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota) \mapsto (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta', \iota)$$

where

$$\begin{aligned}\delta' = \{&(s_{src}, \texttt{true}, \vec{i}', \vec{v}', s_{tgt}, \vec{o}, \vec{a}) \mid (s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta,\\ &(\forall p \in cmp.CPorts_{IN} : \texttt{if } (\vec{i}_p = *) \texttt{ then } \vec{i}'_p \in \vec{I}_p \texttt{ else } \vec{i}'_p = \vec{i}_p),\\ &(\forall var \in cmp.CVars : \texttt{if } (\vec{v}_{var} = *) \texttt{ then } \vec{v}'_{var} \in \vec{V}_{var} \texttt{ else } \vec{v}'_{var} = \vec{v}_{var}),\\ &\phi(\vec{i}, \vec{v})\}.\end{aligned}$$ △

The enabledness expansion function *enablednessExpansion*, as described in Definition 6.24, replaces the $*$ symbol on inputs and variable values on transitions with all possible combinations. An example of the application of the function *enablednessExpansion* is given in the next section as part of a larger example in Figure 6.26. A $^*\text{MAA}_{ts}$ automaton without guard predicates other than **true** and without underspecified input and variable values is not modified by the enabledness expansion from Definition 6.24.

We call a $^*\text{MAA}_{ts}$ automaton with all references removed, without the symbol $*$ on any input port or variable value of any transition, and with all guards set to **true** an enabledness expanded $^*\text{MAA}_{ts}$ automaton. The following definitions of $^*\text{MAA}_{ts}$ semantics all handle enabledness expanded $^*\text{MAA}_{ts}$ automata that are obtained from regular $^*\text{MAA}_{ts}$ automata by applying the function *removeReferences* from Definition 6.22 and the function *enablednessExpansion* from Definition 6.24.

6.4.3. Completions when using $^*\text{MAA}_{ts}$ Automata as Implementations

When using $^*\text{MAA}_{ts}$ automata to implement component behavior, we expect that all relevant information on how a component reacts to its inputs is given in the automaton's transition system. In case no reaction is specified there should be indeed no action taken by the component. In this case the state of the component does not change and the component does not send any messages. In time-synchronous streams the absence of a message is denoted by the symbol ε. The component thus sends the symbol ε on all output ports.

We formally define this implementation-based interpretation of the automaton as ε completion for $^*\text{MAA}_{ts}$ automata. This completion of the transition system removes syntactic-underspecification based on the symbol $*$. It also handles the omission of transitions from $^*\text{MAA}_{ts}$ automata and finally yields a MAA_{ts} automaton. The ε completion is formally defined in Definition 6.25. We define ε completion for enabledness expanded automata, i.e., any $^*\text{MAA}_{ts}$ automaton with the prior application of the functions *removeReferences* from Definition 6.22 and *enablednessExpansion* from Definition 6.24.

All unspecified outputs are replaced by the $\varepsilon_{\lightning}$ message and unassigned variables are assigned their previous values. In case no transition in the ${}^{*}\text{MAA}_{ts}$ automaton is enabled, the $\varepsilon_{\lightning}$-completed automaton sends the $\varepsilon_{\lightning}$ message on all ports and preserves the values of all local variables (see last item in Definition 6.25).

The $\varepsilon_{\lightning}$-completed automaton is a MAA_{ts} automaton that no longer uses the symbol $*$ and has no guards. All occurrences of $*$ are replaced in the first two items of Definition 6.25 and the tuples added to the transition contain only elements from $\vec{I}$, $\vec{V}$, and $\vec{O}$ without the addition of the symbol $*$ as employed in the definition of the structure of ${}^{*}\text{MAA}_{ts}$ automata from Definition 6.19.

Definition 6.25 ($\varepsilon_{\lightning}$ completion for enabledness expanded ${}^{*}\text{MAA}_{ts}$ automata)**.** The $\varepsilon_{\lightning}$ completion takes as input an enabledness expanded ${}^{*}\text{MAA}_{ts}$ automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ and produces the MAA_{ts} automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta', \iota')$, where

1. unspecified initial output is set to $\varepsilon_{\lightning}$:
   $\iota' = \{(s, \vec{o}') \in (S \times \vec{O}) \mid (s, \vec{o}) \in \iota,$
   $\quad \forall p \in cmp.CPorts_{OUT} : \texttt{if } (\vec{o}_p = *) \texttt{ then } \vec{o}'_p = \varepsilon_{\lightning} \texttt{ else } \vec{o}'_p = \vec{o}_p\}$

2. unspecified output is set to $\varepsilon_{\lightning}$ and unspecified variable assignments are set to preserve variable values:
   $\delta'' = \{(s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}', \vec{a}') \mid (s_{src}, \texttt{true}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta,$
   $\quad \forall p \in cmp.CPorts_{OUT} : \texttt{if } (\vec{o}_p = *) \texttt{ then } \vec{o}'_p = \varepsilon_{\lightning} \texttt{ else } \vec{o}'_p = \vec{o}_p,$
   $\quad \forall var \in cmp.CVars : \texttt{if } (\vec{a}_{var} = *) \texttt{ then } \vec{a}'_{var} = \vec{v}_{var} \texttt{ else } \vec{a}'_{var} = \vec{a}_{var}\}$

3. undefined behavior is expanded to preserve the current state and variable values and to produce the symbol $\varepsilon_{\lightning}$ on all output ports:
   $\delta' = \delta'' \cup \{(s_{src}, \vec{i}, \vec{v}, s_{src}, \vec{\varepsilon}_{\lightning}, \vec{v}) \mid \nexists s'_{tgt}, \vec{o}', \vec{a}'.\ (s_{src}, \vec{i}, \vec{v}, s'_{tgt}, \vec{o}', \vec{a}') \in \delta'\}$

△

An example of $\varepsilon_{\lightning}$ completion for a MAA_{ts} automaton is shown in Figure 6.26. The first table shows the single transition from state `idle` to state `driving` of the automaton inside component `BumpControl` shown in Figure 6.21.

The second table shows the transition after the application of the function *enablednessExpansion* from Definition 6.24. The underspecified input on the port `ts` is replaced by the two alternative values $\varepsilon_{\lightning}$ and `ALERT`.

Finally, the third table in Figure 6.26 shows the transition system after $\varepsilon_{\lightning}$ completion defined in Definition 6.25. The previously unhandled input value combinations on the ports `bump` and `ts` are now handled by the automaton. The corresponding outputs are $\varepsilon_{\lightning}$ values on all output ports.

6.4.4. Completions when using MAA_{ts} Automata as Specifications

We define two additional completions that handle syntactic underspecification by replacing the symbol $*$ with multiple behaviors. These completions are useful when using ${}^{*}\text{MAA}_{ts}$ automata as specifications. In these cases we do not expect that the model

Given the following transition from the transition system of the $^*\text{MAA}_{ts}$ automaton shown in Figure 6.21:

s_{src}	$\vec{i}_{bump}$	$\vec{i}_{ts}$	s_{tgt}	$\vec{o}_{rMot}$	$\vec{o}_{lMot}$	$\vec{o}_{tc}$
idle	**true**	*	driving	FORWARD	FORWARD	*

Application of enabledness expansion (Definition 6.24) leads to:

s_{src}	$\vec{i}_{bump}$	$\vec{i}_{ts}$	s_{tgt}	$\vec{o}_{rMot}$	$\vec{o}_{lMot}$	$\vec{o}_{tc}$
idle	**true**	$\varepsilon_{\lightning}$	driving	FORWARD	FORWARD	*
idle	**true**	ALERT	driving	FORWARD	FORWARD	*

$\varepsilon_{\lightning}$ completion (Definition 6.25) leads to:

s_{src}	$\vec{i}_{bump}$	$\vec{i}_{ts}$	s_{tgt}	$\vec{o}_{rMot}$	$\vec{o}_{lMot}$	$\vec{o}_{tc}$
idle	**true**	$\varepsilon_{\lightning}$	driving	FORWARD	FORWARD	$\varepsilon_{\lightning}$
idle	**true**	ALERT	driving	FORWARD	FORWARD	$\varepsilon_{\lightning}$
idle	**false**	$\varepsilon_{\lightning}$	idle	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$
idle	**false**	ALERT	idle	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$
idle	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$	idle	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$
idle	$\varepsilon_{\lightning}$	ALERT	idle	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$	$\varepsilon_{\lightning}$

Figure 6.26.: Example for guards and input expansion and $\varepsilon_{\lightning}$ completion for a single transition of the automaton inside component `BumpControl` (see Figure 6.21).

explicitly contains all reactions of the component specified as transitions. The $^*\text{MAA}_{ts}$ automaton is considered a specification of what the implementation must do.

As an example, consider the specification for component `ToggleSwitch` shown in Figure 6.27. The automaton inside component `ToggleSwitchSpec` requires that every implementation starts with the output **`false`** on the port `active`. When receiving the message **`false`** on the port `pressed` the component has to emit the message **`false`** on the port `active`.

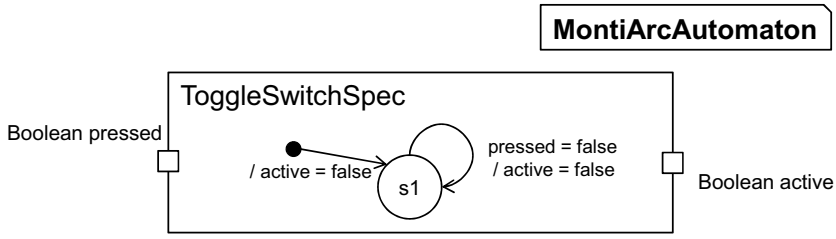


Figure 6.27.: A partial specification for the behavior of component `ToggleSwitch` given as a $^*\text{MAA}_{ts}$ automaton.

When treating the automaton inside component `ToggleSwitchSpec` as an implementation the toggle switch would simply ignore any message not specified, e.g., the message **`true`** on port `pressed`, and send $\varepsilon_\cdot$ messages. Viewed as a specification we want to allow arbitrary behavior if not stated otherwise. This freedom is added to the transition relation of an automaton using chaos completion.

Chaos completion allows an arbitrary reaction to inputs once a component has received an input for which its behavior is undefined. After reacting to the input, the automaton of the component can go to any of its states or a special state s_{chaos} that allows arbitrary behavior. Outputs not specified are interpreted as all possible combinations of outputs. If at least one transition is enabled in the specification automaton the component has to react as specified. Chaos completion has been defined for I/O$^\omega$ automata in [Rum96]. An adapted version for $^*\text{MAA}_{ts}$ automata is given in Definition 6.28.

Definition 6.28 (Chaos completion for enabledness expanded $^*\text{MAA}_{ts}$ automata)**.** The chaos completion takes as input an enabledness expanded $^*\text{MAA}_{ts}$ automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ and produces the MAA_{ts} automaton $(S', \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta', \iota')$, where

1. a chaos state is added:
   $S' = S \cup \{s_{chaos}\}$

2. unspecified initial output is expanded to all possible output combinations:
   $\iota' = \{(s, \vec{o}') \in (S \times \vec{O}) \mid (s, \vec{o}) \in \iota,$
   $\quad \forall p \in cmp.CPorts_{OUT} : \textbf{if } (\vec{o}_p = *) \textbf{ then } \vec{o}'_p \in \vec{O}_p \textbf{ else } \vec{o}'_p = \vec{o}_p\}$

3. unspecified outputs and variable assignments are expanded to all possible combinations of values:
   $\delta'' = \{(s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}', \vec{a}') \in (S' \times \vec{I} \times \vec{V} \times \vec{S'} \times \vec{O} \times \vec{V}) \mid$
   $\quad (s_{src}, \textbf{true}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta,$
   $\quad \forall p \in cmp.CPorts_{OUT} : \textbf{if } (\vec{o}_p = *) \textbf{ then } \vec{o}'_p \in \vec{O}_p \textbf{ else } \vec{o}'_p = \vec{o}_p,$
   $\quad \forall var \in cmp.CVars : \textbf{if } (\vec{a}_{var} = *) \textbf{ then } \vec{a}'_{var} \in \vec{V}_{var} \textbf{ else } \vec{a}'_{var} = \vec{a}_{var}\}$

4. undefined behavior is expanded to all possible behaviors:
   $\delta' = \delta'' \cup \{(s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in (S' \times \vec{I} \times \vec{V} \times \vec{S'} \times \vec{O} \times \vec{V}) \mid$
   $\quad \nexists s'_{tgt}, \vec{o}', \vec{a}' : (s_{src}, \vec{i}, \vec{v}, s'_{tgt}, \vec{o}', \vec{a}') \in \delta''\}$

△

An example for chaos completion of the transition system of the automaton shown in Figure 6.27 is shown in Figure 6.29. Please note that in this case the enabledness expansion from Definition 6.24 has no effect on the transition relation δ since no underspecification based on the symbol $*$ is used in the input or local variables. The transition relation δ is expanded from one tuple to 31 tuples by chaos completion. There are $3 * |S'| * 3$ transitions starting in state s_{chaos} where 3 is the number of distinct messages that can be received on port `pressed` or sent on port `active`. Both ports are of type `Boolean` extended with the special symbol $\varepsilon_\cdot$ (with $|\{\textbf{true}, \textbf{false}, \varepsilon_\cdot\}| = 3$).

Given the transition relation δ of the $^*\text{MAA}_{ts}$ automaton from Figure 6.27:

s_{src}	$\vec{i}_{pressed}$	s_{tgt}	$\vec{o}_{active}$
s1	**false**	s1	**false**

Chaos completion (Definition 6.28) leads to the expanded transition relation with $S' = S \cup \{s_{chaos}\}$:

s_{src}	$\vec{i}_{pressed}$	s_{tgt}	$\vec{o}_{active}$
s1	**false**	s1	**false**
s1	$\varepsilon_{\text{↯}}$	s_{chaos}	**true**
s1	$\varepsilon_{\text{↯}}$	s_{chaos}	**false**
s1	$\varepsilon_{\text{↯}}$	s_{chaos}	$\varepsilon_{\text{↯}}$
s1	$\varepsilon_{\text{↯}}$	s1	**true**
s1	$\varepsilon_{\text{↯}}$	s1	**false**
s1	$\varepsilon_{\text{↯}}$	s1	$\varepsilon_{\text{↯}}$
s1	**true**	$\lvert S' \rvert * 3$ combinations	
s_{chaos}	$3 * \lvert S' \rvert * 3$ combinations		

Figure 6.29.: Chaos completion of the transition system of the automaton inside component `ToggleSwitchSpec` from Figure 6.27.

Chaos completion allows arbitrary behavior once the automaton switches to the chaos state. An alternative to chaos completion is, e.g., relaunch completion without the additional state s_{chaos}. Relaunch completion allows arbitrary behavior in case no transition of the original automaton is enabled but requires a relaunch of the automaton in a state of the automaton after each transition (see relaunch completion for I/O^{ω} automata defined in [Rum96]). One implementation related pattern for MAA_{ts} automata is to idle in a state until a special event/input pattern occurs. When using $^*\text{MAA}_{ts}$ automata as implementations, this behavior is realized by $\varepsilon_{\text{↯}}$ completion. To use this pattern for waiting for special events in specifications we define the output completion for $^*\text{MAA}_{ts}$ automata.

Output completion as defined in Definition 6.30 can be used to specify complex behavior using multiple states with unknown behavior in some or all of the states. This completion is less permissive than chaos completion. If the component receives an input for which no transition of the automaton is activated it can produce an arbitrary output. Similarly, if an output is marked as unspecified using the symbol $*$, the output is arbitrary. The values of local variables are treated as part of the state space of the specification and are thus preserved if not explicitly specified. The execution of the automaton resumes in the state where the previously unhandled input and variable values combination was received.

Definition 6.30 (Output completion for enabledness expanded $^*\text{MAA}_{ts}$ automata)**.** The output completion takes an enabledness expanded $^*\text{MAA}_{ts}$ automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ as input and produces the MAA_{ts} automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta', \iota')$, where

1. unspecified initial output is expanded to all possible output combinations:
   $\iota' = \{(s, \vec{o}') \in (S \times \vec{O}) \mid (s, \vec{o}) \in \iota,$
   $\forall p \in cmp.CPorts_{OUT} : \texttt{if } (\vec{o}_p = *) \texttt{ then } \vec{o}'_p \in \vec{O}_p \texttt{ else } \vec{o}'_p = \vec{o}_p\}$

2. unspecified output is expanded to all possible combinations of values and unspecified variable assignments are set to preserve variable values:
   $\delta'' = \{(s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}', \vec{a}') \mid (s_{src}, \texttt{true}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta,$
   $\forall p \in cmp.CPorts_{OUT} : \texttt{if } (\vec{o}_p = *) \texttt{ then } \vec{o}'_p \in \vec{O}_p \texttt{ else } \vec{o}'_p = \vec{o}_p,$
   $\forall var \in cmp.CVars : \texttt{if } (\vec{a}_{var} = *) \texttt{ then } \vec{a}'_{var} = \vec{v}_{var} \texttt{ else } \vec{a}'_{var} = \vec{a}_{var}\}$

3. undefined behavior is expanded to preserve the current state and variable values and to sending all possible combinations of outputs:
   $\delta' = \delta'' \cup \{(s_{src}, \vec{i}, \vec{v}, s_{src}, \vec{o}, \vec{v}) \mid \nexists s'_{tgt}, \vec{o}', \vec{a}'.\ (s, \vec{i}, \vec{v}, s'_{tgt}, \vec{o}', \vec{a}') \in \delta''\}$

△

The Item 1 of Definition 6.30 adds arbitrary output on the ports with unspecified output (modeled by the symbol $*$). Item 2 of Definition 6.30 adds arbitrary output to all transitions for ports where the output was not specified. The values of local variables are preserved if no assignment was specified. Item 3 of Definition 6.30 adds arbitrary transitions from a state to itself preserving all variable values for all state, input, and variable value combinations that do not enable a transition after the completion of δ to δ'' in Item 2.

An example for output completion of the transition system of the $^*\text{MAA}_{ts}$ automaton shown in Figure 6.27 is shown in Figure 6.31. The transition relation δ is expanded from one tuple to 7 tuples. The 6 additional transitions handle the inputs **true** and ε, for which no transition was defined in the original automaton.

6.4.5. MAA_{ts} Automaton Semantics as I/O Relations and SPF

Our semantics definition for MAA_{ts} automata does not handle arbitrary MAA_{ts} automata as defined in Definition 6.19. For defining its semantics we require that a MAA_{ts} automaton has a defined (possibly non-deterministic) response to every possible input in any possible state and variable configuration. Otherwise, a modeled system would not be able to react to all possible input situations. In our time-synchronous semantics this means that the system would stop time on the output channels. A total transition function ensures that the automaton can answer all requests. A transition function is total if the automaton has at least one enabled transition for every state, input, and valuation of local variables. We call a MAA_{ts} automaton with a total transition relation a total MAA_{ts} automaton (see Definition 6.32).

Given the transition relation δ of the $^*\text{MAA}_{ts}$ automaton from Figure 6.27:

s_{src}	$\vec{i}_{pressed}$	s_{tgt}	$\vec{o}_{active}$
s1	**false**	s1	**false**

Output completion (Definition 6.30) leads to the expanded transition relation:

s_{src}	$\vec{i}_{pressed}$	s_{tgt}	$\vec{o}_{active}$
s1	**false**	s1	**false**
s1	**true**	s1	**true**
s1	**true**	s1	**false**
s1	**true**	s1	$\varepsilon_{\lightning}$
s1	$\varepsilon_{\lightning}$	s1	**true**
s1	$\varepsilon_{\lightning}$	s1	**false**
s1	$\varepsilon_{\lightning}$	s1	$\varepsilon_{\lightning}$

Figure 6.31.: Output completion of the transition system of the automaton inside component `ToggleSwitchSpec` from Figure 6.27.

Definition 6.32 (Total MAA_{ts} automaton)**.** A MAA_{ts} automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ is total if and only if the transition relation contains a transition for every possible state, variable assignment, and input:

$$\begin{aligned}&\forall s_{src} \in S, \vec{i} \in \vec{I}, \vec{v} \in \vec{V} : \\ &\quad \exists s_{tgt} \in S, \vec{o} \in \vec{O}, \vec{a} \in \vec{V} : \\ &\quad\quad (s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta.\end{aligned}$$ △

The values of variables are essentially treated as an extension of the state space of the automaton. Instead of requiring the existence of a transition for every combination of states and variables, we could weaken Definition 6.32 to only require the existence of transitions from reachable states and variable assignments. This modification is not required in our case since the completions $\varepsilon_{\lightning}$ completion (Definition 6.25), chaos completion (Definition 6.28), and reaction completion (Definition 6.30) all result in total automata according to Definition 6.32. Specifically, the transition system of the automata is made total in the last bullet of each of the three definitions.

We now give the semantics of total MAA_{ts} automata as I/O relations. The I/O relation of a MAA_{ts} automaton is the relation of all possible input histories to the respective output histories produced by the automaton. The input histories of a MAA_{ts} automaton $(S, \vec{I}, \vec{V}, \vec{O}, \vec{\gamma}, \delta, \iota)$ are all timed streams $\vec{I}^{\infty}$ over tuples of the input port types of the automaton. The output histories of the automaton are the streams $o \in \vec{O}^{\infty}$ produced by the automaton as a response to an input $i \in \vec{I}^{\infty}$.

A pair consisting of the input streams $i \in \vec{I}^{\infty}$ and the output stream $o \in \vec{O}^{\infty}$ is in the semantics of the automaton if the first element of the output o equals an initial output from the set ι and the input and output streams elements result from tuples in the transition relation of the automaton. We define the I/O relation semantics of total MAA_{ts} automata in Definition 6.33 in two parts. The first part of the definition ensures the existence of an initial output and state pair. The second part of Definition 6.33 introduces a recursive characterization of valid input and output streams based on a state and variable values. This part selects a tuple in δ that produces the current input and output elements. The remainder of the input and output streams is defined by the recursive application of the relation.

Definition 6.33 (I/O relation semantics of total MAA_{ts} automata)**.** The I/O relation semantics of a total automaton $A = (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ is defined as the relation

$$R_A = \{(in, out) \in (\vec{I}^{\infty} \times \vec{O}^{\infty}) \mid \exists (s, \vec{o}) \in \iota : out.0 = \vec{o} \wedge R'_A(s, \gamma, in, rt(out))\}$$

where R'_A is the unique largest solution of the following recursive definition:

$$R'_A = \{(s, \vec{v}, in, out) \in S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty} \mid \exists t \in S, \vec{a} \in \vec{V} : (s, in.0, \vec{v}, t, out.0, \vec{a}) \in \delta \wedge R'_A(t, \vec{a}, rt(in), rt(out))\}$$

△

All reactions to elements of inputs $i \in \vec{I}^{\infty}$ are delayed by one time cycle in the output stream $o \in \vec{O}^{\infty}$ by the first part of Definition 6.33. Since the I/O relation semantics is defined for total MAA_{ts} automata there is at least one pair $(i, o) \in R$ for every $i \in \vec{I}^{\infty}$.

The semantics for total MAA_{ts} automata is well-defined. The recursive definition in the second part of Definition 6.33 has a unique least fixpoint. The functional of the recursive definition is $\sigma : \wp(S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty}) \to \wp(S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty})$ defined as

$$\sigma(X) = \{(s, \vec{v}, in, out) \in S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty} \mid \exists t \in S, \vec{a} \in \vec{V} : (s, in.0, \vec{v}, t, out.0, \vec{a}) \in \delta \wedge X(t, \vec{a}, rt(in), rt(out))\}$$

To show that the recursive definition in Definition 6.33 has a unique solution we follow similar proofs from [Rum96] that show the existence of a unique smallest fixpoint based on the Tarski fixpoint theorem [Tar55]. As in [Rum96], we use the complete partial order of sets where the order $\sqsubseteq$ is defined as the inverse subset relation $S_1 \sqsubseteq S_2 \Leftrightarrow S_2 \subseteq S_1$ with the smallest element S where $\forall S' \in \wp(S) : S \sqsubseteq S'$. We show that the functional σ is monotonic on $\wp(S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty})$ with respect to $\sqsubseteq$, i.e., $\forall X, Y \subseteq S \times \vec{V} \times \vec{I}^{\infty} \times \vec{O}^{\infty} : X \sqsubseteq Y \Rightarrow \sigma(X) \sqsubseteq \sigma(Y)$.

We replace $\sqsubseteq$ with the inverse subset relation $\subseteq$ and unfold the definition of the functional σ for elements in $\sigma(Y)$. We then replace Y with X on the right side, since for $Y \subseteq X$ all $(t, \vec{a}, in, out) \in Y$ are also in X. The right side is now $\sigma(X)$ and thus $X \sqsubseteq Y \Rightarrow \sigma(X) \sqsubseteq \sigma(Y)$:

$$Y \subseteq X \wedge \forall (s, \vec{v}, \vec{i}{:}in, \vec{o}{:}out) \in \sigma(Y) : \exists t \in S, \vec{a} \in \vec{V} : (s, \vec{i}, \vec{v}, t, \vec{o}, \vec{a}) \in \delta \wedge (t, \vec{a}, in, out) \in Y$$
$$\Rightarrow \forall (s, \vec{v}, \vec{i}{:}in, \vec{o}{:}out) \in \sigma(Y) : \exists t \in S, \vec{a} \in \vec{V} : (s, \vec{i}, \vec{v}, t, \vec{o}, \vec{a}) \in \delta \wedge (t, \vec{a}, in, out) \in X$$
$$\Rightarrow \forall (s, \vec{v}, \vec{i}{:}in, \vec{o}{:}out) \in \sigma(Y) : (s, \vec{v}, \vec{i}{:}in, \vec{o}{:}out) \in \sigma(X)$$
$$\Rightarrow \sigma(Y) \subseteq \sigma(X)$$

The functional is monotonic and thus has a unique least fixpoint that uniquely characterizes the largest relation R'_A defined in Definition 6.33.

As an alternative definition of the semantics, we give semantics of total MAA_{ts} as time-synchronous SPF. We define the semantics as a set of time-synchronous SPF by a translation of total MAA_{ts} automata into total time pulsed automata as defined in [Rum96, Definition 5.32]. An adapted structure of time pulsed automata is shown in Definition 6.34. This structure is adapted from [Rum96, Definition 5.32] for the time-synchronous case where each time slice of the input and output streams contains exactly one message. In [Rum96, Definition 5.32] the transition relation is defined to handle time slices of finite streams instead of a single message on all input and output channels.

Definition 6.34 (Time pulsed automaton [Rum96, adapted from Definition 5.32])**.** A time pulsed I/O^{ω} automaton is a tuple $(S, M_{in}, M_{out}, \delta, I)$, where

- S is a non-empty set of states,
- M_{in} is a non-empty input alphabet,
- M_{out} is a non-empty output alphabet,
- $\delta \subseteq S \times M_{in} \times S \times M_{out}$ is a transition relation, and
- $I \subseteq S \times M_{out}$ is a set of initial states and their initial output.

△

The main differences between the structure of time pulsed automata from Definition 6.34 and MAA_{ts} automata from Definition 6.19 are that we allow multiple input and output ports for MAA_{ts} automata and local variables which can be read and set in transitions. We translate these concepts of MAA_{ts} into concepts of time pulsed automata as shown in Definition 6.35. This translation is similar to the translation used in [Rum96, Definition 33] where finite streams on multiple channels are interpreted as a single message in a new alphabet. Here we interpret the messages on all input and output ports as single input and output messages.

Definition 6.35 (Translation of total MAA_{ts} automata to time pulsed automata)**.** The function $toTPA : (S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota) \mapsto (S', M'_{in}, M'_{out}, \delta', I')$ translates a total MAA_{ts} automaton into a total time pulsed automaton, where

- $S' = S \times \vec{V}$
- $M'_{in} = \vec{I}$

- $M'_{out} = \vec{O}$
- $\delta' = \{((s_{src}, \vec{v}), \vec{i}, (s_{tgt}, \vec{a}), \vec{o}) \mid (s_{src}, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta\}$
- $I' = \{((s, \gamma), \vec{o}) \mid (s, \vec{o}) \in \iota\}$

△

A time pulsed automaton $(S, M_{in}, M_{out}, \delta, I)$ is total if and only if it defines at least one transition for every input in every state, i.e., $\forall s \in S, m \in M_{in} : \exists t \in S, o \in M_{out} : (s, m, t, o) \in \delta$ [Rum96, Definition 4.2]. The translation function *toTPA* translates total MAA_{ts} automata into total time pulsed I/O^{ω} automata since the source state and input combinations of the I/O^{ω} automaton are composed of the source state, variable values, and inputs of the total MAA_{ts} automaton.

The semantics of total I/O^{ω} automata as sets of SPF is defined in Definition 6.36. This is a schematic adaption of Proposition 5.34 from [Rum96] to the time-synchronous case. The only changes are again the adaption of the time slices from finite streams to single messages.

Definition 6.36 (Semantics of total time pulsed automata [Rum96, adapted from Proposition 5.34])**.** The set of SPF realized by the total automaton $(S, M_{in}, M_{out}, \delta, I)$ is

$$\begin{aligned}[\![(S, M_{in}, M_{out}, \delta, I)]\!] \quad = \quad & \{g \in M_{in}^{\infty} \to M_{out}^{\infty} \wedge g \text{ strongly causal} \mid \\ & \exists h \in [\![(S, M_{in}, M_{out}, \delta, I)]\!]^{tap}, (s_i, out_i) \in I : \\ & \forall in \in M_{in}^{\infty} : \; g(in) = out_i \frown h(s_i, in)\}\end{aligned}$$

where $[\![.]\!]^{tap}$ is the unique largest solution of the following recursive definition:

$$\begin{aligned}[\![(S, M_{in}, M_{out}, \delta, I)]\!]^{tap} \quad = \quad & \{h \in S \to M_{in}^{\infty} \to M_{out}^{\infty} \wedge \forall s : h(s) \text{ weakly causal} \mid \\ & \forall m \in M_{in}, s \in S : \exists t \in S, out \in M_{out} : (s, m, t, out) \in \delta \wedge \\ & \exists h' \in [\![(S, M_{in}, M_{out}, \delta, I)]\!]^{tap} : \\ & \forall in \in M_{in}^{\infty} : h(s, m \frown in) = out \frown h'(t, in)\}\end{aligned}$$

△

The first part of the definition ensures that every start state and initial output combination of the automaton is covered by at least one SPF. The second part defines a predicate $[\![.]\!]^{tap}$ recursively over the transition relation. This recursive definition is well formed and does have a unique largest solution [Rum96]. The core idea of predicate $[\![.]\!]^{tap}$ is to unfold the transition relation by one step, by defining the state parametrized function h through selection of a transition $(s, m, t, out) \in \delta$ and a continuation function h' that handles the rest. Taking the largest solution resembles that nondeterminism is interpreted as all possible SPF (underspecification).

The translation of MAA_{ts} automata into time pulsed automata and this semantics definition for time pulsed automata immediately gives us a semantics of MAA_{ts} automata as sets of SPF as defined in Definition 6.37.

Definition 6.37 (SPF semantics of total MAA$_{ts}$ automata)**.** The SPF semantics of a total MAA$_{ts}$ automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ is defined as $[\![toTPA(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)]\!]$.

△

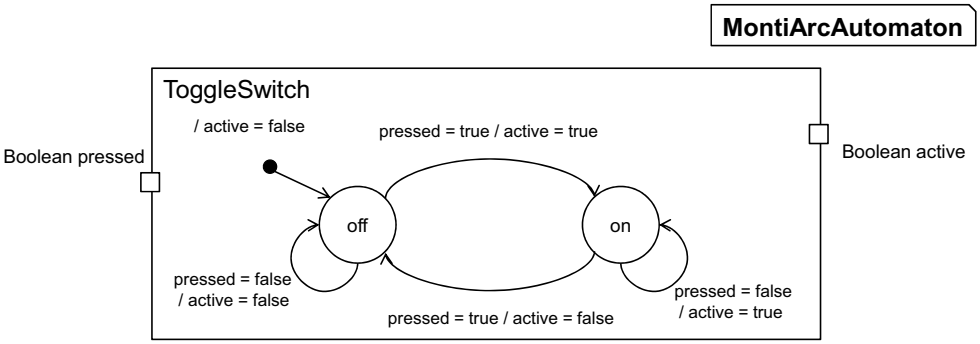


Figure 6.38.: The implementation of component `ToggleSwitch` as a total MAA$_{ts}$ automaton.

As a concrete example consider the automaton inside component `ToggleSwitch` as shown in Figure 6.38. The $\varepsilon_{\text{ɷ}}$ completion of the automaton only adds self-loops to the two states `off` and `on` that take as input $\varepsilon_{\text{ɷ}}$ and produce the output $\varepsilon_{\text{ɷ}}$. The $\varepsilon_{\text{ɷ}}$-completed MAA$_{ts}$ automaton is total according to Definition 6.32, has no guards and all inputs and outputs of each transition are defined. The semantics of the automaton shown in Figure 6.38 is a singleton set consisting of the function g defined by:

$$\begin{aligned} g &: \texttt{Boolean}^{\infty} \to \texttt{Boolean}^{\infty} \\ g(in) &= \textbf{false} \frown h(\texttt{off}, in) \end{aligned}$$

where h is the single element from the set $[\![(S, M_{in}, M_{out}, \delta, I)]\!]^{tap}$ from the second part of Definition 6.36:

$$\begin{aligned} h : \{\texttt{off}, \texttt{on}\} \times \texttt{Boolean}^{\infty} &\to \texttt{Boolean}^{\infty} \\ h(\texttt{off}, \textbf{true} \frown in) &= \textbf{true} \frown h(\texttt{on}, in) \\ h(\texttt{off}, \textbf{false} \frown in) &= \textbf{false} \frown h(\texttt{off}, in) \\ h(\texttt{on}, \textbf{true} \frown in) &= \textbf{false} \frown h(\texttt{off}, in) \\ h(\texttt{on}, \textbf{false} \frown in) &= \textbf{true} \frown h(\texttt{on}, in) \\ h(\texttt{off}, \varepsilon_{\text{ɷ}} \frown in) &= \varepsilon_{\text{ɷ}} \frown h(\texttt{off}, in) \\ h(\texttt{on}, \varepsilon_{\text{ɷ}} \frown in) &= \varepsilon_{\text{ɷ}} \frown h(\texttt{on}, in) \end{aligned}$$

The last line in the definition of the function h is added by the ε completion of the automaton. For total and deterministic time pulsed automata the set of (timed) stream processing functions $[\![(S, M_{in}, M_{out}, \delta, I)]\!]$ is always a singleton [Rum96]. This corresponds to a single possible implementation of the automaton's behavior.

The two semantics for MAA_{ts} automata appear similar but they are different. The functions in the SPF semantics preserves the state changes of a system whereas I/O relation based semantics loses the connection to the states of the automaton. We show in Section 6.5.1 that the two semantics lead to different notions of refinement.

6.5. Refinement of MAA_{ts} Automata

From the definitions of MAA_{ts} automata semantics and the definitions of behavior refinement we immediately receive a refinement for MAA_{ts} automata. In fact, we receive two notions of refinement: one based on I/O relations and one based on SPF. The I/O relation refinement of MAA_{ts} automata is based on the inclusion of input and output pairs of communication histories. All pairs of input and output that the semantics of a more concrete MAA_{ts} automaton allows have to be valid input and output pairs allowed by the semantics of the more abstract MAA_{ts} automaton. The I/O relation refinement for MAA_{ts} automata is given in Definition 6.39.

Definition 6.39 (I/O relation refinement of MAA_{ts} automata)**.** A MAA_{ts} automaton $A_{concrete}$ refines a MAA_{ts} automaton $A_{abstract}$ based on the I/O relation semantics from Definition 6.33 if and only if

$$\forall i \in \vec{I}^{\infty}, o \in \vec{O}^{\infty} : (i, o) \in R_{A_{concrete}} \Rightarrow (i, o) \in R_{A_{abstract}}$$

△

From the semantics definition for MAA_{ts} automata based on SPF (Definition 6.37) we receive a different refinement relation between MAA_{ts} automata. A more concrete MAA_{ts} automaton refines a more abstract MAA_{ts} automaton if all possible realizations of the behavior of the concrete, i.e., all SPF in the automaton's semantics, are also possible realizations of the behavior of the abstract MAA_{ts} automaton. This refinement relation is formally given in Definition 6.40.

Definition 6.40 (SPF refinement of MAA_{ts} automata)**.** A MAA_{ts} automaton $A_{concrete}$ refines a MAA_{ts} automaton $A_{abstract}$ based on the SPF semantics from Definition 6.37 if and only if

$$[\![toTPA(A_{concrete})]\!] \subseteq [\![toTPA(A_{abstract})]\!]$$

△

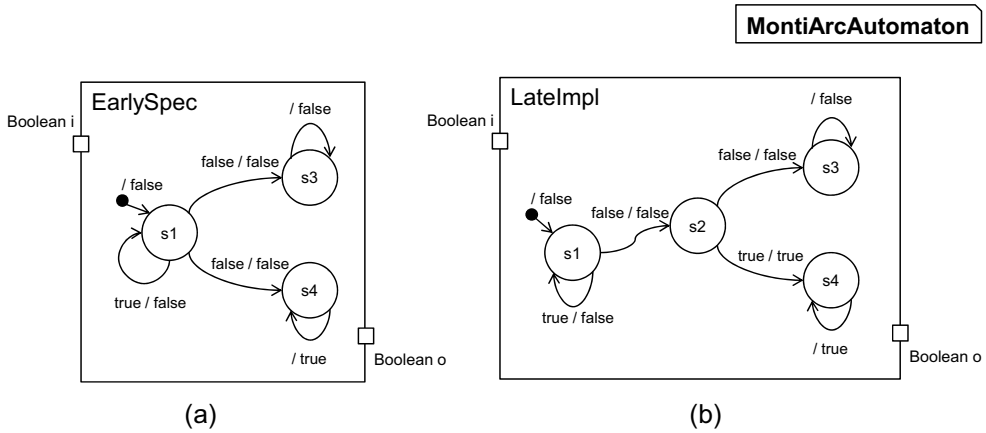


Figure 6.41.: Two MontiArcAutomaton components with automata that demonstrate different semantics of the MAA_{ts} refinement based on the I/O relation semantics and the MAA_{ts} refinement based on the SPF semantics.

6.5.1. Different Refinements

It is important to note that the two notions of refinement based on the different semantics of MAA_{ts} automata are not equivalent. We present an example consisting of two MAA_{ts} automata in Figure 6.41 (a) and (b). The component `EarlySpec` contains a total MAA_{ts} automaton with a single input port and a single output port both of the type `Boolean`. The component `LateImpl` has the same input and output ports.

We show that the component `LateImpl` refines the component `EarlySpec` based on the I/O relation semantics from Definition 6.33 but not based on the SPF semantics from Definition 6.37. Without loss of generality we ignore the symbol ε_{t} and ε_{c} completion in this example: the type `Boolean` could be replaced with a single value taking the role of **`true`** while ε_{t} would take the role of **`false`** in this example.

The automaton of component `EarlySpec` is non-deterministic with a single non-deterministic choice between the transitions to the state `s3` and the state `s4` when receiving the value **`false`** in the state `s1`. The SPF semantics of the automaton is the uncountably infinite set $[\![toTPA(\texttt{EarlySpec})]\!]$ that contains functions $g_K : \texttt{Boolean}^\infty \to \texttt{Boolean}^\infty$ for all $K \subseteq \mathbb{N} \cup \{\infty\}$ where

$$g_K : \begin{cases} \mathbf{true}^n \frown \langle \mathbf{false} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \mapsto \langle \mathbf{false} \rangle \frown \mathbf{false}^n \frown \langle \mathbf{false} \rangle \frown \mathbf{false}^\infty & \text{for } n \in K \\ \mathbf{true}^n \frown \langle \mathbf{false} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \mapsto \langle \mathbf{false} \rangle \frown \mathbf{false}^n \frown \langle \mathbf{false} \rangle \frown \mathbf{true}^\infty & \text{for } n \notin K \end{cases}$$

The inclusion of ∞ in the set K is valid because for a message m the iteration m^∞ is well-defined as the infinite iteration of the message m.

The MAA_{ts} automaton `LateImpl` is total and deterministic. Thus, its semantics given by the set $[\![toTPA(\texttt{LateImpl})]\!]$ contains the single SPF $f : \texttt{Boolean}^\infty \to \texttt{Boolean}^\infty$ as defined below with $n \in \mathbb{N} \cup \{\infty\}$:

$$f : \begin{cases} \mathbf{true}^n \frown \langle \mathbf{false}, \mathbf{false} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \mapsto \\ \qquad\qquad \mathbf{false} \frown \mathbf{false}^n \frown \langle \mathbf{false}, \mathbf{false} \rangle \frown \mathbf{false}^\infty \\ \mathbf{true}^n \frown \langle \mathbf{false}, \mathbf{true} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \mapsto \\ \qquad\qquad \langle \mathbf{false} \rangle \frown \mathbf{false}^n \frown \langle \mathbf{false}, \mathbf{true} \rangle \frown \mathbf{true}^\infty \end{cases}$$

The function f does not equal any function g_K for $K \subseteq \mathbb{N}$. Assume $\exists K \subseteq \mathbb{N} : g_K = f$. From $f(\langle \mathbf{false} \rangle \frown \mathbf{true}^\infty) = \langle \mathbf{false}, \mathbf{false} \rangle \frown \mathbf{true}^\infty = g(\langle \mathbf{false} \rangle \frown \mathbf{true}^\infty)$ it follows from the definition of g_K that $0 \in K$. From $f(\mathbf{false}^\infty) = \mathbf{false}^\infty = g(\mathbf{false}^\infty)$ it follows from the definition of g_K that $0 \notin K$. We thus have a contradiction and it follows that $\forall K \subseteq \mathbb{N} : g_K \neq f$. In summary, the automaton `LateImpl` does not refine the automaton `EarlySpec` with the SPF semantics since $[\![toTPA(\texttt{LateImpl})]\!] \notin [\![toTPA(\texttt{EarlySpec})]\!]$.

The I/O relation semantics MAA_{ts} automaton `LateImpl` is the relation $R_{\texttt{LateImpl}}$ where

$$\begin{aligned} R_{\texttt{LateImpl}} = \{(i, o) \in \texttt{Boolean}^\infty \times \texttt{Boolean}^\infty \mid \\ \exists n \in \mathbb{N} \cup \{\infty\} : \\ (i = \mathbf{true}^n \frown \langle \mathbf{false}, \mathbf{false} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \wedge o = \mathbf{false}^\infty) \vee \\ (i = \mathbf{true}^n \frown \langle \mathbf{false}, \mathbf{true} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty \wedge o = \mathbf{false}^{n+2} \frown \mathbf{true}^\infty)\} \end{aligned}$$

Both cases of the I/O tuples in the I/O relation $R_{\texttt{LateImpl}}$ are also contained in the I/O relation semantics of the MAA_{ts} automaton `EarlySpec`: for all $n \in \mathbb{N} \cup \{\infty\}$ the I/O tuples $(\mathbf{true}^n \frown \langle \mathbf{false}_{\texttt{s3}}, \mathbf{false} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty, \mathbf{false}^\infty)$ are contained in the relation $R_{\texttt{EarlySpec}}$. The subscript `s3` for the message **false** denotes the non-deterministic choice of the target state `s3`, which leads to the output $\mathbf{false}^\infty$. Similarly, for all $n \in \mathbb{N} \cup \{\infty\}$ the I/O tuples $(\mathbf{true}^n \frown \langle \mathbf{false}_{\texttt{s4}}, \mathbf{true} \rangle \frown \{\mathbf{true}, \mathbf{false}\}^\infty, \mathbf{false}^{n+2} \frown \mathbf{true}^\infty)$ are contained in the relation $R_{\texttt{EarlySpec}}$. The subscript `s4` for the message **false** denotes the non-deterministic choice of the target state `s4`, which leads to the output suffix $\mathbf{true}^\infty$.

Thus, the different semantics lead to different refinements.

6.6. Related Work

In this section we discuss related types of automata and some works that deal with incompleteness and uncertainty in automata models. We also compare MontiArcAutomaton to approaches that go beyond the modeling of automata and include component models as does MontiArcAutomaton.

6.6.1. Related Types of Automata

Mealy automata [Mea55] and Moore automata [Moo56] are finite state automata with finite input and output alphabets. Mealy and Moore automata consume input words and produce output words. Transitions in Mealy automata are labeled with one input and one output letter. Transitions in Moore automata are only labeled with an input letter and each state is labeled with one output letter.

I/O automata by Lynch and Tuttle [LT89] are a description mechanism for describing components that are executed in parallel and communicate asynchronously by executing actions. The actions of I/O automata are divided in input, output, and internal actions. All actions are atomic. Every transition of an I/O automaton is labeled with one action. Transitions labeled with input actions can be executed when the identical action is executed as an output action by another automaton. The composition of multiple I/O automata is a synchronization of the execution of transitions of composed automata on actions with the same name. This requires, e.g., the disjointness of the sets of output actions of different automata because only one component should control the execution of each action.

Message-passing automata [BL01, BL06] are an automaton model that describes communicating components. A single message passing automaton consists of a finite set of local automata that represent components. Components are connected pairwise by reliable channels of unbounded size that allow an asynchronous FIFO communication [BL06]. Because of the unbounded channel size and asynchronous communication the model is more expressive than MAA_{ts} automata. The modeling language MontiArcAutomaton however may also describing message-passing automata as presented in [BL01, BL06].

Constraint automata [BSAR06] are an extension of I/O automata. Constraint automata were introduced as a semantic model of connectors in the connector coordination framework Reo [Arb04]. The transitions of constraint automata are guarded with sets of active channels and constraints over the data on these channels. Baier et al. [BSAR06] relate constraint automata to a timed data stream semantic defined for Reo connectors in [AR02] and define refinement (containment) of constraint automata as the containment of relations over timed data streams in the semantics of the constraint automata. Constraint automata and their semantics are somewhat related to the automata of MontiArcAutomaton because of the timed data streams semantics and the concept of channels. However, constraint automata focus on the description of connectors rather than that of components. A combination of these complementing approaches might be an interesting future work.

Alfaro and Henzinger [dAH01] introduced interface automata to describe possible compositions of components. Interface automata capture input assumptions and output guarantees of components. A set of components can be composed if there is at least one environment that fulfills the assumptions of the composition. The structure of interface automata is similar to I/O automata. The assumptions of an automaton to the environment are that the environment provides only inputs the automaton can handle and that it accepts all outputs the automaton provides. In Focus [BS01] and for the MontiArcAutomaton components defined in Section 6.4 all components (including the

environment) are considered to be input-enabled (total), i.e., accepting all inputs in any reachable state. The correctness of component composition in our case is thus only a syntactic type compatibility criterion for connected ports. Extensions of FOCUS to partial behavior, which could benefit from using composition techniques similar to the one for interface automata [dAH01], exist [Bro05] but are outside the scope of this work.

Other language profiles of MontiArcAutomaton

The MAA_{ts} language profile is only one profile of the modeling language MontiArcAutomaton. Other profiles put different restrictions on the abstract syntax of automata and define different semantics. Our initial works were based on a similar profile for time-synchronous communication without the symbol ε, for the absence of messages [Kir11]. This profile has a time-synchronous semantics but requires the models to receive and send messages in every time step, i.e., when waiting for an event all cases and time steps of the event not arriving have to be handled explicitly by additional transitions.

In another work on MontiArcAutomaton we have experimented with a language profile for single event-based processing of messages [Mar12]. In this language profile the input on transitions is restricted to a single input on a single port while output is allowed on all output ports and with an arbitrary finite stream of events. The communication in this work is not time-synchronous.

More general, the modeling language MontiArcAutomaton is a member of the MontiArc [HRR12] language family. MontiArc is a modeling language for modeling C&C systems. The modeling language itself does not include concepts for the implementation of component behavior. We have added these concepts in MontiArcAutomaton by embedding automata to describe component behavior as described in Chapter 6 and [RRW14]. Another modeling language of the MontiArc family is AJava [HRR10]. AJava embeds elements of the programming language Java in MontiArc component definitions.

Underspecification mechanisms and refinement in automata

An important property of behavioral specifications is to allow underspecification and refinement, i.e., to leave open some details in a specification and provide them later. A specification mechanism typically used to express underspecification is non-determinism available in most automata specification languages. We review some additional mechanisms below.

An interesting difference between interface automata [dAH01] and MAA_{ts} automata is the definition of refinement. One motivation for different refinement is that interface automata are not input-enabled while MAA_{ts} automata with their FOCUS semantics are, i.e., components in FOCUS produce output histories for all input histories. We thus use a refinement based on containment and I/O stream relations while interface automata use an alternating simulation [AHKV98] refinement. This means that the more concrete automaton can simulate all input steps of the abstract one while the abstract can simulate all output steps of the concrete, i.e., an implementation may allow more inputs and less outputs.

MontiArcAutomaton allows the creation of *partial models* of a behavior in the sense that required behavior is explicitly modeled whereas unknown behavior is not. Other approaches to modeling possible and required behavior are modal transition systems (MTS) [LT88, Lar89] where behavior expressed by a set of transitions is marked as *may* or *must*. Disjunctive MTS (DMTS) [LX90] extend MTS with disjunctions between *must* transitions outgoing from a common state.

Informally, a refining MTS has to preserve all *must* transitions of the MTS it refines. The refining MTS can preserve *may* transitions, change them to *must* transitions, or remove them. For DMTS only one *must* transition of each disjunction has to be preserved. Refinement relations for MTS are typically defined based on similarity relations between states. For classic strong modal refinement [LT88] the corresponding states of the refining MTS can simulate all *must* transitions and the states of the more abstract MTS can simulate all *may* transitions. Other refinement relations are, e.g., syntactic refinement where the relation between the states of the MTSs is a partial injective function from the states of the more concrete to the states of the abstract MTS [LNW07b] and weak modal refinement where the *must* and *may* transitions can be simulated with additional internal actions of the simulating MTS [LNW07a].

As a major difference to MAA_{ts} automata the various notions of refinement of MTS are all contravariant: allowed behavior must have been allowed and required behavior is still required. Our semantics of MAA_{ts} automata does not explicitly distinguish between *may* an *must* and it does allow the removal of behavior as long as input-enabledness of MAA_{ts} automata is preserved. Finally, in MTS and DMTS all behaviors have to be modeled explicitly while MontiArcAutomaton offers implicit completions.

Famelis et al. [FBDCS11] have introduced partial modeling and a language independent refinement for partial models [SFC12]. Partial models extend a modeling language, e.g., with markers for optional or abstract elements. The refinement introduced in [SFC12] is purely syntactic based on the introduced annotations. It thus does not take into account the semantics of a model. As an example of a difference, our approach defines refinement for MontiArcAutomaton automata based on their I/O behavior. One automaton may refine a syntactically very different automaton that it does not necessarily refine syntactically.

6.6.2. State-Based System Modeling Languages

Statecharts by Harel [HP85, Har87] are one of the most prominent visual description techniques for reactive systems. Statecharts combine many modeling language features, e.g., hierarchical states, action execution in states, receiving and sending signals. After their introduction many variants [Bee94] were derived and statecharts were adopted as part of the standardized UML [Obj12a]. Scholz [Sch98] has defined a semantics based on message streams for a subset of Harel's statecharts with a refinement that corresponds to our definition in Section 6.3.4. A different refinement in the context of object oriented systems has been defined by Harel and Kupferman [HK02]. The refinement is based on the object oriented `is-a` relation and thus requires that a refined statechart has every behavior the abstract statechart has. The refinement thus may only add behavior while

we consider refinements that remove behavior (remove uncertainty).

Some of the features available in statecharts, e.g., hierarchical states, are likely to be also useful for the modeling language MontiArcAutomaton. However, we have no sufficient experience yet how decomposition of components relates to decomposition of states and what is more useful in the development of interactive systems. We leave these evaluations and possible modeling language extensions as future work.

Statecharts are standardized both in UML [Obj12a] and SysML [Obj12b]. As described before these languages also offer the modeling of component types and C&C models (see Section 2.4). The combination of statecharts with SysML's internal block diagrams or UML's composite structure diagrams allows similar models to the models we describe using MontiArcAutomaton. We consider it an advantage of MontiArcAutomaton to allow the definition of a component and its behavior in one artifact without additional bindings and allocation as necessary in SysML.

The AutoFOCUS tool [HSSS96, HS97, BHS99, HF07] for the specification and prototyping of distributed systems allows the behavioral specification of components using state transition diagrams. These can be edited in a graphical representation and translated into executable simulation code. Currently, AutoFOCUS supports time-synchronous streams with weakly and strongly causal behavior of components. The automata for modeling AutoFOCUS components are similar to MAA_{ts} automata. In contrast to MontiArcAutomaton AutoFOCUS does not distinguish between a component and its type and it does not support the instantiation of components.

The block diagram language implemented in MathWorks Simulink [wwwn] is extended with state transition diagrams in Stateflow [wwwo]. The automata of Stateflow are a combination of Mealy and Moore machines and they are fully integrated in the simulation and code generation environment of Stateflow. The semantics of Stateflow diagrams is only given informally but has been formalized in many ways by various translations into other formalisms [MC12]. To the best of our knowledge there is no support for underspecification and refinement of Stateflow automata as allowed in MAA_{ts} automata (see Section 6.4).

Ptolemy II [Pto14, wwwaa] is a modeling and simulation tool for actor systems. Ptolemy allows the composition of models with heterogeneous models of computation (semantics domains and scheduling) [Lee10]. The discrete event and the synchronous-reactive model of computation are similar to the FOCUS time-synchronous streams of our semantics [TSSL13]. In the description of Ptolemy II [Pto14] the term refinement refers to the syntactical hierarchical decomposition of state machines. We are not aware of refinement and specification mechanisms in Ptolemy II similar to completions of MAA_{ts} automata.

Finally, many works consider hybrid systems and extensions of automata with multiple clocks, differential equations, and probabilities [Rab63, AD94, GSB98, LSV03]. These extensions go beyond the scope our current work.

Chapter 7.

An Analysis Framework for Component Behavior

The development of the functional behavior of component and connector systems typically starts with initial specifications of the behavior of a system or its subsystems. These initial models contain underspecified behavior, which leaves some decisions to a more detailed later specification or an implementation. Adding details to specifications is known as refinement. A refinement preserves or removes possible behaviors from the specification of a system.

In this chapter we address the challenge of automatically verifying the behavior refinement and equality relations between MontiArcAutomaton models. By extending the notion of equality and refinement to behavior observed on shared ports, our work allows to define behavior specifications only for relevant ports.

Engineers may define refinement and equality checks for component type definitions in MontiArcAutomaton specification suites. We present a modeling language for MontiArcAutomaton specification suites and checks. The definition and organization of specification checks in MontiArcAutomaton specification suites allows hiding technical details from the users of the framework.

To solve the analysis problems we express the semantics of MontiArcAutomaton component type definitions as Mona [EKM98, wwwz] programs. Our translation supports composed component type definitions as well as different completions of automata. Each translated component type definition results in a predicate over input and output histories of the component. Due to this general translation, the analysis opportunities provided by this general translation are not limited to checking equivalence and refinement.

We have reported on the modeling of functional requirements using MontiArcAutomaton automata in [RRW12] and our code generation approach for MontiArcAutomaton models has been presented in [RRW13b].

Chapter outline and contributions

We present an example for the application of the refinement of component behavior in Section 7.1 and formulate the MontiArcAutomaton component behavior refinement and equality analysis problems in Section 7.2.

As one of the main contributions of this chapter we express the semantics of MontiArcAutomaton component type definitions as Mona [EKM98, wwwz] programs in Section 7.3 providing full automation for solving the analysis problems. We present a modeling language for MontiArcAutomaton specification suites and specification checks for conveniently expressing the MontiArcAutomaton behavior analysis problems in Section 7.4. Section 7.5 provides additional analysis examples and their formulation as MontiArcAutomaton specification suites.

Section 7.6 gives an overview of our implementation and evaluation. We discuss the presented approach in Section 7.7 and conclude this chapter with an overview of related work in Section 7.8.

7.1. Specification and Analysis Example

We present an example for the development of a software controller for the bumper bot shown in Figure 7.1. We have introduced parts of the example system in Section 6.1. The main objective of the bumper bot is to drive around and traverse a floor. In case the robot bumps into an obstacle it should back up, turn to another direction, and continue the exploration of the floor.

Figure 7.1.: The bumper bot robot with a touch sensor in front and two motors to power the left and right wheels.

The structure of the system architecture of the robot is shown in Figure 7.2. The parent component `SimpleBumperBot` is marked with the stereotype «`deploy`» since this component aggregates the components deployed to the robots physical control device. The parametrized component `TouchSensor` provides access to the values read on the touch sensor mounted at the front of the robot shown in Figure 7.1. The two components `mRight` and `mLeft` of the parametrized component type `Motor` provide access to the physical motors powering the left and right wheels as shown in Figure 7.1. The component `BumpControl` shown in the center of Figure 7.2 controls the robot.

The complete interface of the component `BumpControl` is shown in Figure 7.3. The ports `bump` and `ts` (on the left) are input ports and ports `lMot`, `rMot`, and `tc` (on the

right) are output ports.

The components `TouchSensor`, `Timer`, and `Motor` are reused from an available component library. It is now the task of the engineers to develop the MontiArcAutomaton implementation of the component `BumpControl` of the robot architecture shown in Figure 7.2.

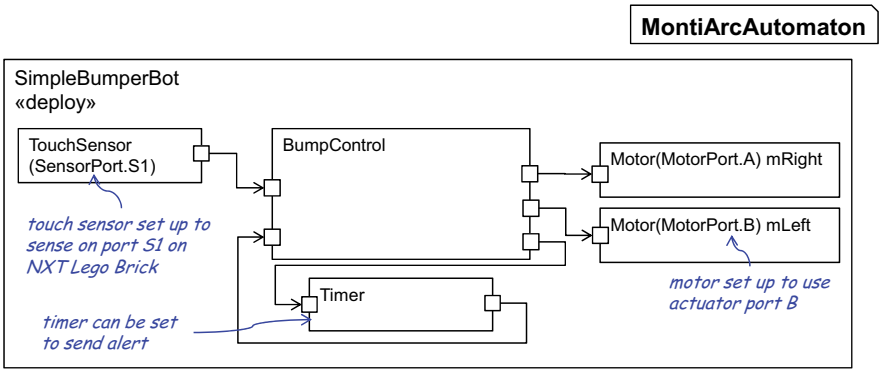


Figure 7.2.: The architecture definition `SimpleBumperBot` of the bumper bot robot. The top level component is marked with the stereotype «`deploy`». It describes the deployment of the displayed components on the physical control device of the robot shown in Figure 7.1.

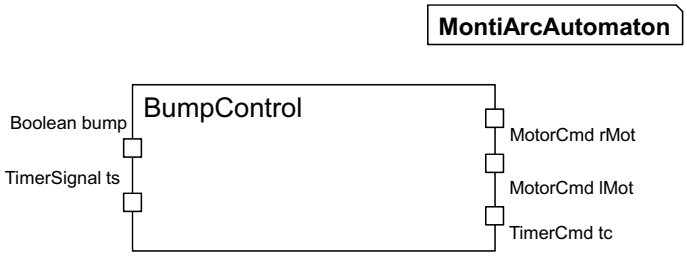


Figure 7.3.: The complete interface of component `BumpControl`. The input ports of the component are shown on the left and the output ports are shown on the right.

7.1.1. Specification of Required Component Behavior

In a first step an engineer has formulated the requirement [Spec1a] that the robot should not move forward until the bumper has been pressed to initially activate the robot.

- [Spec1a] When turned on, the bumper bot does not drive until the bumper is pressed.

The engineer translated this requirement into the MontiArcAutomaton specification `BumpControlSpec1a` shown in Figure 7.4. She decided to apply chaos completion to the automaton (see Definition 6.28) and accordingly marked the component with the stereotype «`chaosCompletion`». Please note that this specification does not mention the input port `ts` or the output port `tc` of the component `BumpControl` shown in Figure 7.3. These ports are not relevant for the specification.

The concrete syntax expression `--` denotes the absence of a message and corresponds to the symbol ε, used in the semantics definition of ${}^{*}\mathrm{MAA}_{ts}$ automata. The specification `BumpControlSpec1a` only allows behaviors, where for $k \in \mathbb{N}$ the input **`false`**^{k} on the port `bump` implies that at any time $t \leq k+1$ there is either no output on the ports `rMot` and `lMot` or the output is the message `STOP` on both ports. The specification leaves underspecified what happens if the value **`true`** is received on the input port `bump`. In this case the specification allows arbitrary behavior because of chaos completion.

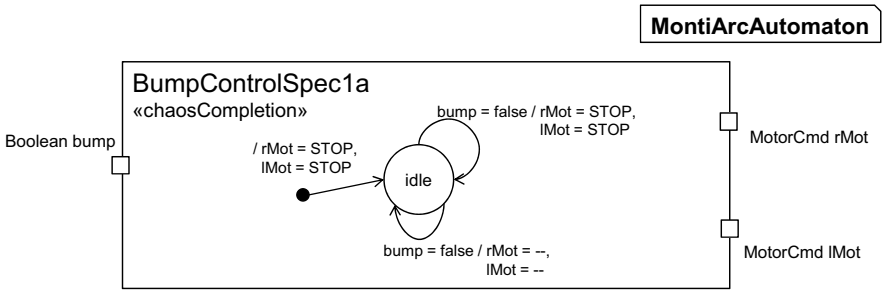


Figure 7.4.: The MontiArcAutomaton specification of the behavior of component `BumpControl`.

During a discussion, the engineer extended the initial specification to specification [Spec1b] as follows:

- [Spec1b] When turned on the bumper bot does not drive until the bumper is pressed. The bumper bot starts going forward if the bumper is pressed after being turned on.

The updated MontiArcAutomaton specification `BumpControlSpec1b` is shown in Figure 7.5. The transition leaving the state `idle` to the state `driving` is enabled when the bump sensor is pressed. The specification determines the possible transitions to choose from in state `idle` for the two expected inputs **`true`** and **`false`** on the input port `bump`. The automaton does not define a transition for the case that no message is received on the port `bump`. The specified behavior is unconstrained after the specification enters the state `driving`.

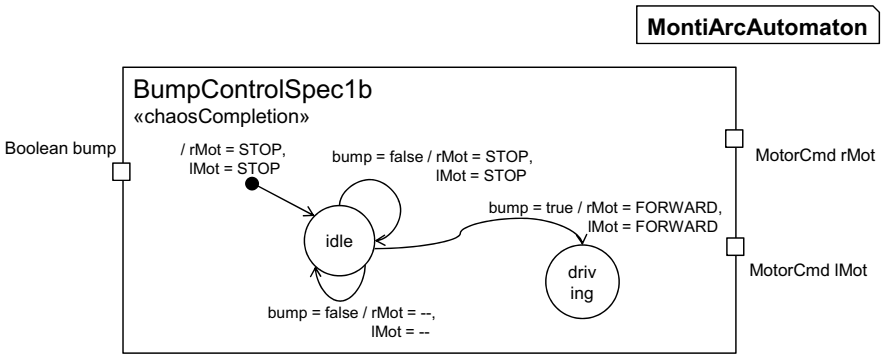


Figure 7.5.: A MontiArcAutomaton specification expressing [Spec1b]. This specification refines `BumpControlSpec1a`.

The engineer makes sure that `BumpControlSpec1b` refines `BumpControlSpec1a`. In addition the engineer can later check that the implementation of component `BumpControl` from Section 6.1 is a refinement of both specifications.

Consider the MontiArcAutomaton specification `BumpControlSpec1c` shown in Figure 7.6, which also expresses the requirements of [Spec1a] and [Spec1b]. This specification is a refinement of the component `BumpControlSpec1a` and the component `BumpControlSpec1b`. The difference between the two automata in Figure 7.5 and Figure 7.6 is the additional state `driving` which is not added in the `BumpControlSpec1c`. Through chaos completion of the transitions of this state the specification `BumpControlSpec1b` allows arbitrary behavior after the initial $\mathbf{false}^k \frown \langle \mathbf{true} \rangle$ sequence on the input port `bump`. On the other hand, the specification `BumpControlSpec1c` requires that pressing the bump sensor always results in sending a forward command to both motors. For the initial activation of the robot this behavior is reasonable. Later on, a pressed bump sensor signals a collision with another object and driving forward is no longer the desired behavior for the bumper bot. Although the specification refines previous specifications and expresses desired requirements, it is too restrictive.

7.1.2. Specifications for Composed Components

Another engineer is working on an extended bumper bot robot with an emergency stop switch as shown in Figure 7.7.

The component architecture of the robot is shown in Figure 7.8. The two sensor reading components for the emergency switch and the front bumper are arranged on the left while the components to control the two motors are arranged on the right. The central component of the robot architecture is the component `BumpControlES`.

The emergency switch is constructed as a touch sensor mounted on top of the robot (see Figure 7.7). The sensor is read by the component `ToggleSensor` which is decomposed (see, e.g., Figure 6.9) into a component of the type `TouchSensor` and a component of

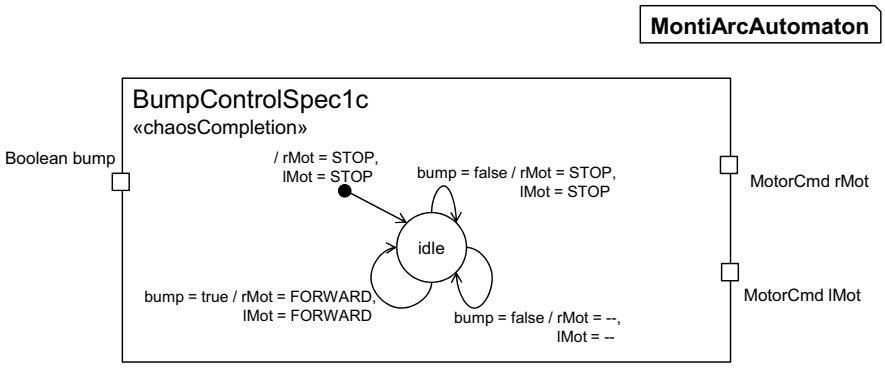


Figure 7.6.: A MontiArcAutomaton specification expressing [Spec1c]. This specification refines `BumpControlSpec1a` and `BumpControlSpec1b`.

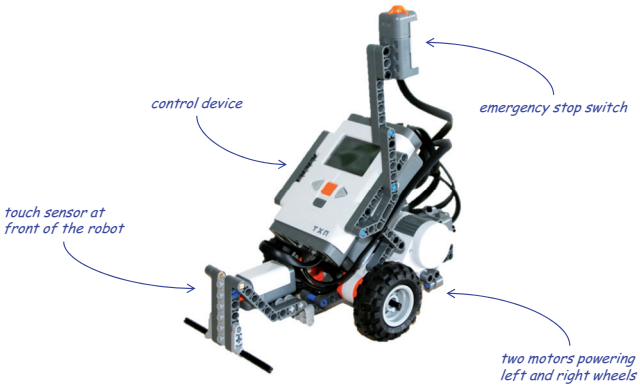


Figure 7.7.: A picture of the extended bumper bot with an emergency stop switch.

the type `ToggleSwitch` (shown in Figure 6.38).

A requirement for the behavior implementation of the component `RobotControllerES` is given as [Spec2].

- [Spec2] While the emergency stop switch is pressed the robot stops the engines.

The engineer expresses the requirement as an automaton in Figure 7.9 inside component `RobotControllerESSpec2`. The specification of the behavior is not complete. Instead of using chaos completion, the engineer decided to use output completion (see Definition 6.30) for the transition system of the automaton. Output completion does not add additional states or transitions between states. Thus, every time the value `true` is received on the port `emgStp`, the `STOP` command has to be sent on both output ports to the motors. For all other inputs the specification allows arbitrary outputs.

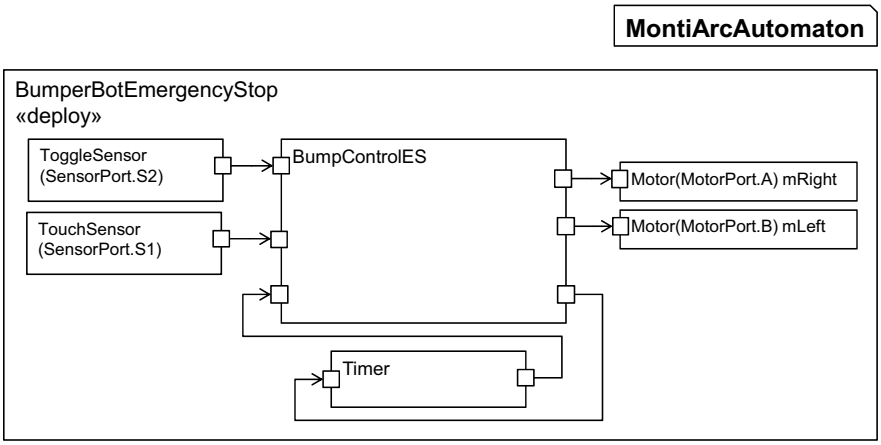


Figure 7.8.: The architecture of of the extended bumper bot with an emergency stop switch. Component `BumpControlES` is composed from the components `MotorStopper`, `Arbiter`, and `BumpControl` as shown in Figure 7.10.

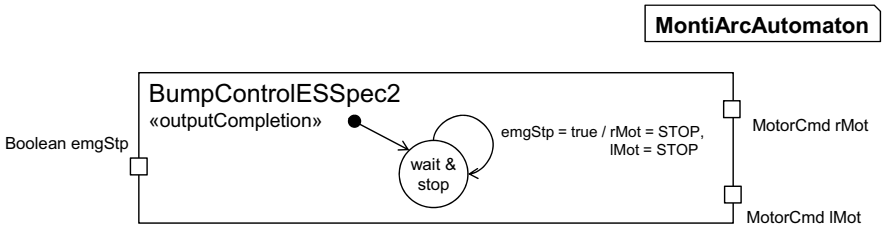


Figure 7.9.: Specification [Spec2] expressed as an automaton.

The engineer working on the robot with the emergency stop switch later created an implementation of the component `BumpControlES` based on some components previously created by other engineers and components taken from a component library. The composed component `BumpControlES` is shown in Figure 7.10. The subcomponent `BumpControl` was introduced in the example in Section 6.1, Figure 6.3 and Section 7.1.1.

The component `MotorStop` is a library component. It constantly sends the message `STOP` of the type `MotorCommand` defined in the class diagram shown in Figure 6.4. The engineer has also developed the subcomponent `ArbiterMotorCmd` shown in Figure 7.11. It has two pairs of input ports of the type `MotorCmd`, one control port of the type `Boolean` with the name `mode`, and one pair of output ports again of the type `MotorCmd`. Based on the Boolean value received on its port `mode` the arbiter forwards the first or second pair of inputs to its output ports. Specifically, if the emergency button is toggled the arbiter receives the value **`true`** on the port `mode` and forwards the constant

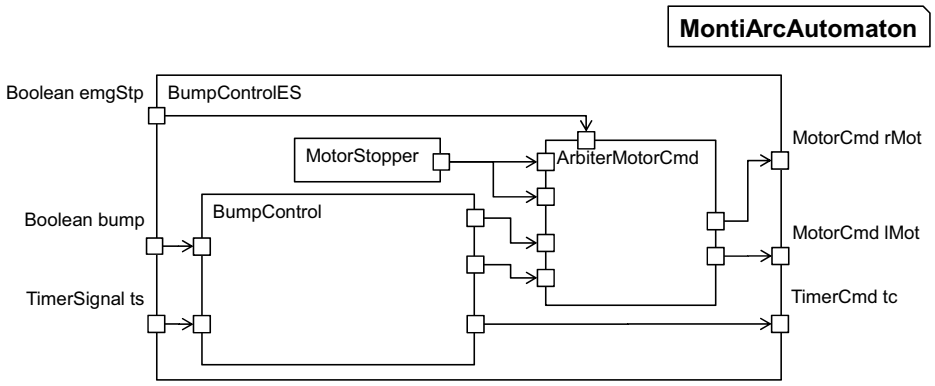


Figure 7.10.: Component `BumpControlES` is composed of components `MotorStopper`, `Arbiter`, and `BumpControl`.

output `STOP` of `MotorStopper` to both motors of the robot.

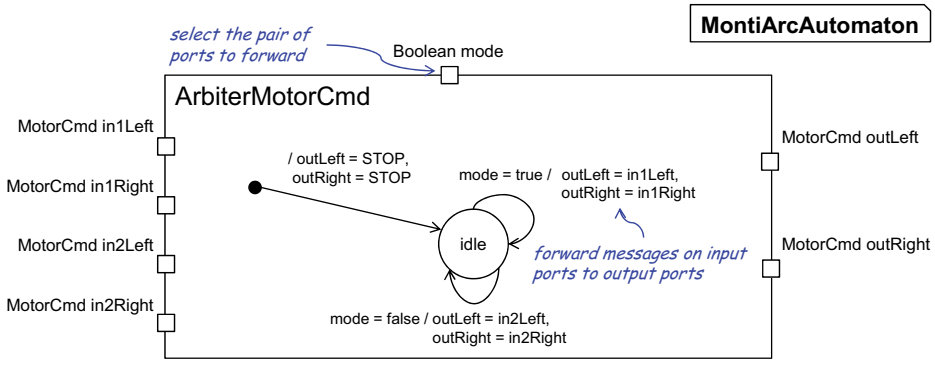


Figure 7.11.: The component `ArbiterMotorCmd` from a component library implemented as a $^*\text{MAA}_{ts}$ automaton. The outputs on the transitions of the state `idle` refer to input port names and thus forward the corresponding input message as output.

After creating the implementation, the engineer successfully validates that the composed robot controller `BumpControlES` refines the specification `BumpControlESSpec2`.

7.2. Behavior Refinement and Equality Analysis Problem

From the definition of the semantics of MAA_{ts} automata in Definition 6.33 and the semantics of composed component types from Definition 6.15 we obtain I/O relation

semantics definitions for decomposed as well as for atomic MontiArcAutomaton component type definitions. In the following definition we thus refer to the semantics of component types as their relations $R \subseteq \vec{I}^{\infty} \times \vec{O}^{\infty}$.

7.2.1. MontiArcAutomaton Component Refinement

The definition of upward simulation refinement for I/O relation semantics is given in Section 6.3.4. We are interested in upward simulation refinement of components where an implementation *impl* may have additional inputs and additional outputs compared to the specification *spec*, that it refines. Formally, this is denoted $spec.CPorts_{IN} \subseteq impl.CPorts_{IN}$ and $spec.CPorts_{OUT} \subseteq impl.CPorts_{OUT}$. The complete input for the component *impl* is the tuple of streams $in \in \bigtimes_{p \in impl.CPorts_{IN}} (p.type^{\infty})$. We denote the restriction of the input *in* of the component *impl* to the input ports of the component *spec* as $in|_{spec.CPorts_{IN}} = \bigtimes_{p \in spec.CPorts_{IN}} (in_p)$ where in_p selects the stream on the input port p. The restriction of component output is defined analogously.

We formally define MontiArcAutomaton component behavior refinement as I/O relation refinement with upward simulation in Definition 7.12.

Definition 7.12 (MontiArcAutomaton component behavior upward simulation refinement)**.** For two MontiArcAutomaton component type definitions $spec, impl \in CTDefs$ with $spec.CPorts_{IN} \subseteq impl.CPorts_{IN}$ and $spec.CPorts_{OUT} \subseteq impl.CPorts_{OUT}$ the component type *impl* is an upward simulation behavior refinement of the component type *spec* if and only if

$$\forall i \in \bigtimes_{p \in impl.CPorts_{IN}} (p.type^{\infty}), o \in \bigtimes_{p \in impl.CPorts_{OUT}} (p.type^{\infty}) : \\ R_{impl}(i, o) \Rightarrow R_{spec}(i|_{spec.CPorts_{IN}}, o|_{spec.CPorts_{OUT}})$$ △

The MontiArcAutomaton refinement problem is to decide for two component type definitions *spec* and *impl* whether *impl* is a component behavior refinement of *spec* according to Definition 7.12. Please note that this notion of refinement requires that the implementation refines the specification in all possible environments, i.e., independent of the inputs on the ports $impl.CPorts_{IN} \setminus spec.CPorts_{IN}$.

The definition of refinement in Definition 7.12 allows the specification of behavior refinement on a subset of component interfaces. In some cases it is also interesting to check a refinement relation that allows the substitution of the specification *spec* by the implementation *impl* [BS01, LS11]. This requires that all inputs required by the component type definition *impl* are also provided for *spec*, i.e., $impl.CPorts_{IN} \subseteq spec.CPorts_{IN}$, and that all outputs provided by *spec* are also provided by *impl*, i.e., $spec.CPorts_{OUT} \subseteq impl.CPorts_{OUT}$.

We formally define MontiArcAutomaton component behavior refinement as I/O relation refinement in Definition 7.13.

Definition 7.13 (MontiArcAutomaton component behavior refinement)**.** For two MontiArcAutomaton component type definitions $spec, impl \in CTDefs$ with $impl.CPorts_{IN} \subseteq spec.CPorts_{IN}$ and $spec.CPorts_{OUT} \subseteq impl.CPorts_{OUT}$ the component type *impl* is a behavior refinement of the component type *spec* if and only if

$$\forall i \in \times_{p \in spec.CPorts_{IN}}(p.type^{\infty}), o \in \times_{p \in impl.CPorts_{OUT}}(p.type^{\infty}) : \\ R_{impl}(i|_{impl.CPorts_{IN}}, o) \Rightarrow R_{spec}(i, o|_{spec.CPorts_{OUT}}) \quad \triangle$$

7.2.2. MontiArcAutomaton Component Equality

We define MontiArcAutomaton component behavior equality as the equality of the streams on shared input and output ports. The restriction to shared ports is more permissive than upward simulation but it ensures symmetry of the equality defined in Definition 7.14.

Definition 7.14 (MontiArcAutomaton component shared ports behavior equality)**.** Two MontiArcAutomaton component type definitions $impl_1, impl_2 \in CTDefs$ have equal I/O behavior on shared ports if and only if

$$\forall i \in \times_{p \in impl_1.CPorts_{IN} \cup impl_2.CPorts_{IN}}(p.type^{\infty}), \\ o \in \times_{p \in impl_1.CPorts_{OUT} \cup impl_2.CPorts_{OUT}}(p.type^{\infty}) : \\ R_{impl_1}(i|_{impl_1.CPorts_{IN}}, o|_{impl_1.CPorts_{OUT}}) \Leftrightarrow \\ R_{impl_2}(i|_{impl_2.CPorts_{IN}}, o|_{impl_2.CPorts_{OUT}}) \quad \triangle$$

The MontiArcAutomaton component equality problem is to decide for two component type definitions $impl_1$ and $impl_2$ whether the two have equal component behavior on shared ports according to Definition 7.14.

For components with different interfaces the equality of Definition 7.14 does not necessarily allow the substitution of a component type definition by one with equal behavior on shared ports. It can however easily be strengthened by the requirement for equal interfaces of the component type definitions as defined in Definition 7.15. The additional requirement of equal component interfaces can easily be checked on the component type definition syntax.

Definition 7.15 (MontiArcAutomaton component behavior equality)**.** Two MontiArcAutomaton component type definitions $impl_1, impl_2 \in CTDefs$ with equal interfaces $impl_1.CPorts = impl_2.CPorts$ have equal I/O behavior if and only if

$$\forall i \in \times_{p \in impl_1.CPorts_{IN}}(p.type^{\infty}), o \in \times_{p \in impl_1.CPorts_{OUT}}(p.type^{\infty}) : \\ R_{impl_1}(i, o) \Leftrightarrow R_{impl_2}(i, o) \quad \triangle$$

7.2.3. Component Specification Analysis in Mona

The analysis problems, definitions, and structures presented so far operate on infinite timed streams. We will present a representation of streams in Mona in Section 7.3.2 on which we base our analysis. This representation is limited to finite streams.

A well known result in model checking is that for two finite transition systems the inclusion of finite traces is equivalent to the inclusion of finite and infinite traces [BK08, Theorem 3.30]. We can employ this result to show that refinement and equality for all finite I/O histories of finite MAA_{ts} automata is equivalent to refinement and equality on infinite streams. Transitions systems from [BK08, Definition 2.1] as used in the proof by Baier and Katoen are constructed from a MAA_{ts} automaton $(S, \vec{I}, \vec{O}, \vec{V}, \vec{\gamma}, \delta, \iota)$ where

- $S' = \delta \cup \iota$ is the finite set of states,
- $I' = \iota$ is the finite set of initial states,
- $AP' = \vec{I} \times \vec{O}$ is the finite set of labels,
- $L'(r) = \begin{cases} (\vec{i}, \vec{o}) \text{ for } r = (s, \vec{i}, \vec{v}, t, \vec{o}, \vec{v}) \in \delta \\ (\varepsilon, \vec{o}) \text{ otherwise} \end{cases}$ is the labeling function, and
- $Post'(r) = \begin{cases} \{(s, \vec{i}, \vec{v}, t, \vec{o}, \vec{v}) \in \delta \mid r = (s, \vec{o}') \wedge v = \vec{\gamma}\} \text{ for } r \in \iota \\ \{(s, \vec{i}, \vec{v}, t, \vec{o}, \vec{a}) \in \delta \mid r = (t, \vec{i}', \vec{a}, t', \vec{o}', \vec{v}')\} \text{ otherwise} \end{cases}$ is the successor function of the transition system.

Please note that the proof of [BK08, Theorem 3.30] relies on the finiteness of the set of states and the set of labels. Thus, it only holds for MAA_{ts} automata with finite port types, finite variable types, and a finite set of transitions.

7.3. Translation into Mona

We present a solution to solving the refinement analysis problem for MontiArcAutomaton components by a reduction to an analysis problem in a decidable fragment of second order logic. We have implemented a translation of $^*\text{MAA}_{ts}$ automata into Mona programs [EKM98] that can be analyzed by the Mona model checker [wwwz]. We introduce Mona in Section 7.3.1 and give an overview of our representation of streams in Mona in Section 7.3.2.

The modeling languages MontiArc and MontiArcAutomaton distinguish between two types of components: atomic and composed. The behavior of a composed component is defined as the composition of the behavior of its subcomponents. This composition is oblivious to whether a subcomponent is atomic or again composed. We preserve this feature of hiding the implementation details of components in our translation to Mona and handle the translation of composed components separately from atomic components while ensuring a common interface that hides implementation details. This separation allows us to extend the analysis framework to components implemented in other modeling languages integrated in MontiArc or MontiArcAutomaton, e.g., using the I/O tables behavior modeling language introduced in [RRW13c]. It also allows the creation of manual implementations for components in Mona as presented in Section 7.3.5.

We show a translation of composed components from Definition 6.8 in Section 7.3.3 and a translation of atomic components with $^*\text{MAA}_{ts}$ automata from Definition 6.20 in Section 7.3.4. The current translation only handles $^*\text{MAA}_{ts}$ automata without guards and with port and variable types that have finite sets of values. Currently supported types are the type `Boolean` and enumerations from UML/P class diagrams. The values of an enumeration type are its enumeration constants.

7.3.1. Mona and the WS1S Logic

Second order logic is first order logic extended with quantification over predicates. Weak monadic second order logic of one successor (WS1S) is a logic with quantification over finite sets of elements with one successor. The word *monadic* restricts the quantified predicates to be unary, i.e., sets, and the word *weak* in the name of the logic restricts sets to be finite. These restrictions make WS1S a decidable logic [Tho90]. The complexity of the decision procedure is non-elementary in the length of the WS1S formula [Mey75].

Mona [EKM98, wwwz] is an implementation of a decision procedure of WS1S and related logics. The basic elements in WS1S and Mona are elements from propositional logic, e.g., conjunction, disjunction, and implication, elements from first order logic (natural numbers), and second order elements (finite sets). In addition to the basic elements Mona adds syntactic sugar and allows, e.g., the addition of constant integer values to first order elements (multiple applications of the successor function `+1`).

The basic syntax of Mona that we will use in this work is shown in Figure 7.16. Mona allows quantification over 0^{th}, 1^{st}, and 2^{nd} order elements using the universal quantifier `all0`, `all1`, and `all2` and the existential quantifiers `ex0`, `ex1`, and `ex2`. Mona also defines the usual operators for sets, e.g., intersection, union, or the subset relation.

Mona Syntax	**Explanation**
`all0`, `all1`, `all2`	universal quantification over propositional values (0^{th} order), natural numbers (1^{st} order), and sets (2^{nd} order)
`ex0`, `ex1`, `ex2`	existential quantification
`&`	conjunction
`|`	disjunction
`<=>`	equivalence of Boolean values(0^{th} order)
`=`	equality (for 1^{st}, and 2^{nd} order elements)
`+1`	successor function for 1^{st} order elements
`empty`	empty set (2^{nd} order)
`inter`	intersection of sets of natural numbers (2^{nd} order)
`union`	union of sets of natural number (2^{nd} order)
`sub`	subset relation (2^{nd} order)

Figure 7.16.: Syntactic elements of WS1S in Mona

A Mona program may consist of variable declarations, predicate definitions, assertions of Boolean statements, and Boolean statements that evaluate to **`true`** or **`false`**. The example Mona program shown in Listing 7.17 is based on the specification of the behavior of component `ToggleSwitch` from the example in Section 6.3.2. The Mona program declares two variables `pressed` and `active` as sets of natural numbers. The set `pressed` is fixed by the `assert` statement to contain only the natural numbers 2 and 5 (Listing 7.17, l. 4). The expression shown in lines 6 to 9 is a schematic translation

of the FOCUS specification shown in Figure 6.12:

$$active.0 = \textbf{false} \ \wedge \forall n \in \mathbb{N} : pressed.n = \textbf{true} \Leftrightarrow active.(n+1) = \neg active.n$$

Since the input stream of the component `ToggleSwitch` is of the type `Boolean`, we represent it in Listing 7.17 as a set of natural numbers where the stream contains the value **true** at time $t \in \mathbb{N}$ if and only if $t \in$ `pressed`.

Mona

```
1 var2 pressed;
2 var2 active;
3
4 assert {2,5} = pressed;
5
6 0 notin active &
7   all1 t:
8     t in pressed <=>
9       (t+1 in active <=> t notin active);
```

Listing 7.17: A specification of the behavior of the component `ToggleSwitch` based on the FOCUS specification from Figure 6.12.

Running the Mona program in Listing 7.17 results in checking the satisfiability of the expression in lines 6-9 under the assumption that the expression in line 4 holds for all possible values of the variables `pressed` and `active`.

Running a Mona program in general may have three different kinds of outputs:

tautology The formula is valid for all possible assignments to free variables. Mona computes a smallest example that satisfies the formula.

contradiction The formula is unsatisfiable. Mona computes a smallest example that does not satisfy the formula.

examples The formula is neither a tautology nor a contradiction. Mona computes an assignment that satisfies the formula and one that does not.

The example computed by Mona for the program shown in Listing 7.17 is depicted in Figure 7.18. The computed example corresponds to the streams shown in Section 6.3.2:

$$pressed = \langle \textbf{false}, \textbf{false}, \textbf{true}, \textbf{false}, \textbf{false}, \textbf{true} \rangle \frown \textbf{false}^{\infty}$$
$$active = \langle \textbf{false}, \textbf{false}, \textbf{false}, \textbf{true}, \textbf{true}, \textbf{true}, \textbf{false} \rangle \frown \textbf{false}^{\infty}$$

It is important to note that Mona can only handle finite sets and thus only finite streams. One problem of the simple translation scheme applied in Listing 7.17 is that for all times `t` we specify properties for times `t + 1`. In case the input stream *pressed* would contain the message **true** an odd number of times (the toggle switch is not

MonaAnalysisOutput

```
1 A satisfying example of least length (6) is:
2 pressed         X 001001
3 active          X 000111
4
5 pressed = {2,5}
6 active = {3,4,5}
```

Listing 7.18: An example for an assignment that satisfies the formula given in the Mona program from Listing 7.17 as computed by Mona.

turned off) the specification becomes unsatisfiable since the set `active` would need to be infinite.

We extend the simple encoding of streams of the type `Boolean` to streams of other finite types and address the problem of infinitely many executions of transitions in our translation of ${}^*\text{MAA}_{ts}$ automata into Mona.

7.3.2. Streams and Stream Processing in Mona

As we have seen in the example presented in Listing 7.17, sequences of messages may be encoded as sets of natural numbers. To express streams of finite message types we use an encoding similar to the one applied by Schätz [Sch09]. For each message on a stream we create one set of natural numbers. As an example, consider the enumeration type `MotorCmd` with the three enumeration values `STOP`, `FORWARD`, and `BACKWARD` from the initial example shown in Figure 6.4. To encode a stream $rightMotor \in \texttt{MotorCmd}^*$ we declare three second order variables in Mona as shown in Listing 7.19, ll. 1-3, one for each possible value `STOP`, `FORWARD`, and `BACKWARD`.

It is necessary to ensure that at each point in time the stream will not have more than one value, i.e., that each natural number $t \in \mathbb{N}$ is contained in only one of the sets. Thus, we assert that the intersection of each set representing a possible value on the stream with the union of all other sets is empty. Asserting a predicate in a Mona program ensures that the analysis only considers assignments where the predicate holds. The assertions are shown in Listing 7.19, ll. 6-11. This part of the translation requires one assertion for every possible value of the stream's type.

Finally, we have to assert that at every point in time one value is defined for the stream, i.e., the union of the sets representing the values on the stream at any point in time contains all points in time represented by the set `allTime`. This assertion is shown in Listing 7.19, ll. 14-15.

The special set `allTime` is defined as the set containing all natural numbers up to some finite point in time. We define `allTime` only once. When declaring more than one stream the common set `allTime` makes sure the streams have the same length. The declaration of the set `allTime` is shown in Listing 7.20. The direction of the implication $\forall t \in \mathbb{N} : t + 1 \in \texttt{allTime} \Rightarrow t \in \texttt{allTime}$ in line 3 is very important. The implication $\forall t \in \mathbb{N} : t \in \texttt{allTime} \Rightarrow t + 1 \in \texttt{allTime}$ would be too restrictive since there is only

Mona

```
1  var2 rightMotor_STOP;
2  var2 rightMotor_FORWARD;
3  var2 rightMotor_BACKWARD;
4
5  # at most one value at any time
6  assert rightMotor_STOP inter
7    (rightMotor_FORWARD union rightMotor_BACKWARD) = empty;
8  assert rightMotor_FORWARD inter
9    (rightMotor_STOP union rightMotor_BACKWARD) = empty;
10 assert rightMotor_BACKWARD inter
11   (rightMotor_STOP union rightMotor_FORWARD) = empty;
12
13 # at least one value at any time
14 assert (rightMotor_STOP union rightMotor_FORWARD
15                         union rightMotor_BACKWARD) = allTime;
```

Listing 7.19: The stream *rightMotor* ∈ MotorCmd* encoded in Mona using the variable allTime that defines all points in time of interest (see Listing 7.20).

one finite set that satisfies it: the empty set. Please note that the set allTime is of arbitrary finite size.

Mona

```
1 # global system time
2 var2 allTime; # universal quantification over all sets allTime
3 assert all1 t: t+1 in allTime => t in allTime;
```

Listing 7.20: The set all time containing all points in time.

The concrete stream *rightMotor* = ⟨STOP, STOP, FORWARD, BACKWARD, STOP, STOP⟩ is encoded using the four sets as shown in Table 7.21. In this example, the set allTime contains all natural numbers from 0 to 5. Thus, we have a finite prefix of the timed stream *rightMotor* up to time 5.

For the translation of *MAA$_{ts}$ automata into Mona we need to add the special value ε$_t$ to all types. We represent the value ε$_t$ at time t as the absence of t from all sets defining the values of the stream. Thus, to add the value ε$_t$ to the type of a stream we replace the assertion of at least one value at any time in allTime by a weaker assertion. Replacing the equality in line 15 of Listing 7.19 by the subset relation sub restricts all streams (possibly containing the special symbol ε$_t$) to the points in time defined by allTime.

The I/O relation semantics of MontiArcAutomaton components are relations over input and output streams. To encode the I/O relation of a stream processing component in Mona we encode the I/O relation as a Mona predicate over input and output streams. An example of a Mona predicate over the input stream *pressed* ∈ Boolean* and the output stream *active* ∈ Boolean* of the component ToggleSwitch is shown in List-

Mona 2^{nd} order variable	value
rightMotor_STOP	$\{0, 1, 4, 5\}$
rightMotor_FORWARD	$\{2\}$
rightMotor_BACKWARD	$\{3\}$
allTime	$\{0, 1, 2, 3, 4, 5\}$

Table 7.21.: The stream $rightMotor = \langle \texttt{STOP}, \texttt{STOP}, \texttt{FORWARD}, \texttt{BACKWARD}, \texttt{STOP}, \texttt{STOP} \rangle$ in its Mona representation.

ing 7.22. The declaration of a Mona predicate starts with the keyword `pred` and the name of the predicate. The head of the predicate also contains its parameters enclosed in parenthesis as shown in lines 1-2 of Listing 7.22. The body of the predicate is an expression that either evaluates to **`true`** or **`false`**. Head and body of a predicate are connected with the symbol =.

Mona

```
1 pred ToggleSwitch(var2 pressed_false, var2 pressed_true,
2   var2 active_false, var2 active_true)
3   = 0 in active_false &
4   # further expressions here
5   ;
```

Listing 7.22: A Mona predicate of the behavior of the component `ToggleSwitch` based on our encoding of streams and the FOCUS specification shown in Figure 6.12.

With a translation of MontiArcAutomaton component type definitions into parametrized Mona predicates the head of the predicate only depends on the interface of the component. Thus, we achieve a compositional and uniform translation. As a result we benefit from the concept of information hiding of component type definitions.

7.3.3. Translation of Composed Components to Mona

Our translation of composed component type definitions follows the semantics definition in Definition 6.15. The body of a predicate representing a composed component is an instantiation of the predicates of the component's subcomponents with suitable parameters.

We briefly review the four kinds of connectors in composed component type definitions from Definition 6.8 to explain the naming conventions of Mona variables in our translation:

parent-to-parent an input port of the parent component is directly connected to one of its output ports,

parent-to-child an input port of the parent component is connected to an input port of a subcomponent,

child-to-parent the output port of a subcomponent is connected to an output port of the parent component, and

child-to-child the output port of a subcomponent is connected to the input port of another subcomponent.

In all four cases a source port may be connected to multiple target ports. For parent-to-parent and parent-to-child connectors the source port is a port of the parent component. The variables that represent the stream on an input port are declared in the signature of the predicate. The name of each variable is the name of the component *cType* and the name of the input port.

For child-to-parent connectors variables for the streams on the target ports are again declared in the signature of the Mona predicate. To uniformly reference variables we change the name of the variable declared in the head of the Mona predicate to the name of the source component and source port of the child-to-parent connector (see the equality in Definition 6.15, Item 3).

For child-to-child connectors we introduce a set of existentially quantified variables representing the stream on the internal channel with the name of the source component and port (unless this variable already exists in the head of the predicate).

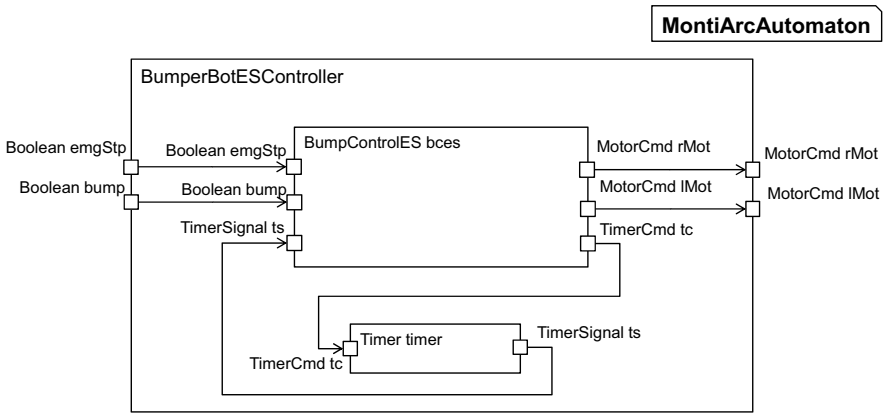


Figure 7.23.: The composed component `BumperBotESController` with its subcomponents `BumpControlES` and `Timer`.

Consider the component `BumperBotESController` shown in Figure 7.23 with its subcomponents `BumpControlES` and `Timer`. This composed component is translated into a predicate with the signature $\Phi_{\texttt{BumperBotESController}} : (\texttt{Boolean}^* \times \texttt{Boolean}^* \times \texttt{MotorCmd}^* \times \texttt{MotorCmd}^*) \rightarrow \mathbb{B}$. According to the naming conventions of variables for streams in Mona the predicate of the composed component `BumperBotESController`

is defined as:

$$\Phi_{\texttt{BumperBotESController}}(s_{\texttt{BumperBotESController\_emgStp}}, s_{\texttt{BumperBotESController\_bump}}, s_{\texttt{bces\_rMot}}, s_{\texttt{bces\_lMot}}) \Leftrightarrow$$
$$\exists s_{\texttt{bces\_tc}} \in \texttt{TimerCmd}^*, s_{\texttt{timer\_ts}} \in \texttt{TimerSignal}^* :$$
$$\Phi_{\texttt{BumpControlES}}(s_{\texttt{BumperBotESController\_emgStp}}, s_{\texttt{BumperBotESController\_bump}}, s_{\texttt{Timer\_ts}}, s_{\texttt{bces\_rMot}}, s_{\texttt{bces\_lMot}}, s_{\texttt{bces\_tc}}) \wedge$$
$$\Phi_{\texttt{Timer}}(s_{\texttt{bces\_tc}}, s_{\texttt{timer\_ts}})$$

Please note that in the translation into Mona each stream is represented by a set of 2^{nd} order variables as described in Section 7.3.2.

Translation rules for composed components

We now formally define the translation rules for composed component type definitions from Definition 6.8 to a Mona predicate. The translation follows the definition of the semantics of composed component types in Definition 6.15. An overview of the translation rules is shown in Figure 7.24. A complete example for the translation of a composed component is included in Appendix L.1.

Translation Rule

Overview of the translation of a composed component type definition into Mona:

```
# predicate for component cType
executeRule(C1) see Figure 7.25 for body of rule C1
# connectors/local channels of component
executeRule(C2) see Figure 7.26 for body of rule C2
# parent-to-parent connectors
executeRule(C3) see Figure 7.28 for body of rule C3
# instantiation an connection of subcomponents on channels
executeRule(C4) see Figure 7.29 for body of rule C4
```

Figure 7.24.: Overview of the translation of composed component types into the concrete syntax of the Mona language. The comments above the rule execution commands belong to the respective rules and are reproduced in this listing to give an intuition of the rules' contents.

The translation of composed components has four parts. The translation rule C1 creates the head of the Mona predicate representing the component. The translation rule C2 declares existentially quantified variables for the streams on local channels, the trans-

lation rule C3 handles input that is directly forwarded as output by asserting equality of the corresponding parameters of the predicate, and the translation rule C4 instantiates the parametrized predicates of the subcomponents of the composed component.

The translation rules operate on the abstract syntax of component type definitions, e.g., the line $\forall sub \in CSubCmps$ would execute the following indented lines for each subcomponent in the set $CSubCmps$. The result produced in the target language (here the Mona language) is marked by `underlining the elements` of the target language syntax. Explanations of the execution of translation rules are given in Appendix B. We provide concrete examples in the figures of most of the rules referenced in Figure 7.24, e.g., for rule C1 in Figure 7.25.

The translation rule C1 is shown in Figure 7.25. The rule translates the type definition of a composed component (Definition 6.8) into the head of a parametrized Mona predicate. The parameters of the predicate are the input and output message streams represented in Mona according to the name pattern introduced in Section 7.3.2. The first part of the rule declares a predicate with the name of the component type $cType$ from Definition 6.8 to uniquely identify the predicate. First, the rule iterates over all input ports and creates parameters for the predicate that are Mona 2^{nd} order variables representing input streams. The result of the application of the rule to the component `BumperBotESController` is shown in the lower part of Figure 7.25.

The second set of variables — declared in the iteration over all output ports of the component type definition — represents the streams on the output ports of the component. For each of the output ports the rule determines the unique connector that connects to the output port. The application of the definite description operator *THE* always yields a unique result because every outgoing port of a composed component type definition has one child-to-parent or parent-to-parent connector (Definition 6.8, Item 11). In the case that the output port is the target port of a child-to-parent connector the variables are named after the source subcomponent and port. Otherwise, the variables of the output streams are named after the component type and output port.

In the latter case of a parent-to-parent connector the streams on the connected input and output ports are required to be equal. The equality constraint for the Mona variables representing the stream on the input port and the stream on the output port is added by translation rule C3 presented in Figure 7.28.

The translation rule C2 is shown in Figure 7.26. The rule adds variable declarations for the local channels of the component, i.e., variables for streams on child-to-child connectors.

The set $locChs$ consists of all output ports of subcomponents minus the ports connected to output ports of the parent component (kinds child-to-parent and parent-to-parent). The latter are subtracted because the variables for the streams on the source ports of child-to-parent connectors are declared by the translation rule C1 in the head of the predicate.

The translation rule C2 shown in Figure 7.26 iterates over the component and port pairs in $locChs$ and declares variables for streams in Mona that are prefixed with the name of the subcomponent and its output port. Finally, the translation rule ensures

Translation Rule

Translation rule:

C1 `# predicate for component` *cType*
`pred` *cType* `(`
$\forall p \in CPorts_{IN}$: *declare variables for all input streams*
$\forall val \in p.type$:
`var2` *cType*`_`*p.name*`_`*val*`,` **-to-parent connector source*
$\forall p \in CPorts_{OUT}$: *variables for output streams named after their source*
let *con* = *THE con* $\in CCons$: $con.tgtCmp = cType \wedge con.tgtPort = p$ **in**
$\forall val \in con.srcPort.type$:
if(*con.srcCmp* $\neq$ *cType***) then** *child-to-parent connector*
`var2` *con.srcCmp*`_`*con.srcPort.name*`_`*val*`,`
else *parent-to-parent connector*
`var2` *cType*`_`*p.name*`_`*val*`,`
`# common time range allTime`
`var2 allTime )`

Result of application to composed component type definition `BumperBotES-Controller` from Figure 7.23:

Mona

```
1 # predicate for component BumperBotESController
2 pred BumperBotESController (
3   var2 BumperBotESController_emgStp_false, var2
       BumperBotESController_emgStp_true,
4   var2 BumperBotESController_bump_false, var2
       BumperBotESController_bump_true,
5   var2 bces_rMot_FORWARD, var2 bces_rMot_BACKWARD, var2
       bces_rMot_STOP,
6   var2 bces_lMot_FORWARD, var2 bces_lMot_BACKWARD, var2
       bces_lMot_STOP,
7   # common time range allTime
8   var2 allTime )
```

Figure 7.25.: Translation rule C1 for the translation of the interface of composed component type definitions into the head of a Mona predicate.

that each set of variables representing a stream conforms to the constraints for streams in Mona by executing the translation rule STRM for each stream.

The parametrized translation rule STRM is shown in Figure 7.27. The rule introduces constraints for the Mona variables used to represent streams. The parameters of the

Translation Rule

Translation rule:

C2 `# connectors/local channels of component`

let $locChs = \{(srcCmp.name, srcPort) \mid$ *subcomponents' output ports* $srcCmp \in CSubCmps \wedge srcPort \in srcCmp.CPorts_{OUT}\} \setminus$
$\{(con.srcCmp, con.srcPort) \mid$ *output ports connected to parent* $con \in CCons \wedge con.tgtCmp = cType\}$ **in**

ex2 { *iterate over output ports of subcomponents in locChs*

$\{srcCmp\_srcPort.name\_val\}'_{val \in srcPort.type}$

$\}'_{(srcCmp,srcPort) \in locChs}$:

`# values on internal channels`

$\forall (srcCmp, srcPort) \in locChs$:

executeRule(STRM $srcCmp\_srcPort.name$, $srcPort.type$**)**

see Figure 7.27 for body of rule STRM

Result of application to composed component type definition `BumperBotES-Controller` from Figure 7.23:

Mona

```
1 # connectors/local channels of component
2 = ex2 bces_tc_DELAY, bces_tc_DOUBLE_DELAY, bces_tc_ABORT,
    timer_ts_ALERT:
3   rule STRM applied to bces_tc and type TimerCmd
4   rule STRM applied to timer_ts and type TimerSignal
```

Figure 7.26.: Translation rule C2 for the translation of child-to-child connectors in composed component type definitions to variables in Mona.

translation rule are the name *strmName* of the stream and its type given as the set of values *Univ*. The translation rule STRM from Figure 7.27 consists of two parts. The first part introduces the constraint that the stream defines only messages at the points in time considered (points in time from the set `allTime`). The second part of rule STRM adds the constraint that only one value (possibly ε) is defined on the stream for all points in time.

The translation rule C3 is shown in Figure 7.28. This rule handles the special case of a parent-to-parent connectors. For parent-to-parent connectors the translation rule C1 from Figure 7.25 declares two streams each with the name of the input port and the name of the output port of the component, respectively. The translation rule C3 presented in Figure 7.28 implements the parent-to-parent connector by stating the equality of all variables representing the input and the output streams on the connected ports.

The translation rule C4 is shown in Figure 7.29. This rule iterates over all subcom-

Translation Rule

Translation rule with parameters $strmName$ and $Univ$:

STRM $\underbrace{\texttt{\# defined values in all points in time}}$

$\{\!|strmName\_val|\!\}^{\overbrace{\texttt{union}}}_{val \in Univ}$ $\underbrace{\texttt{sub allTime \&}}$

$\underbrace{\texttt{\# at most one value on stream at any point in time}}$

$\forall v_1 \in Univ:$

$strmName\_v_1$ $\underbrace{\texttt{inter (}}$ $\{\!|strmName\_v_2|\!\}^{\overbrace{\texttt{union}}}_{v_2 \neq v_1 \in Univ}$ $\underbrace{\texttt{) = empty \&}}$

Result of application to parameters $strmName$ = $\underbrace{\texttt{bces\_tc}}$ and $Univ$ = {DELAY, DOUBLE_DELAY, ABORT}:

Mona

```
1 # defined values in all points in time
2 bces_tc_DELAY union bces_tc_DOUBLE_DELAY union bces_tc_ABORT sub
      allTime &
3 # at most one value on stream at any point in time
4 bces_tc_DELAY inter (bces_tc_DOUBLE_DELAY union bces_tc_ABORT )
     = empty &
5 bces_tc_DOUBLE_DELAY inter (bces_tc_DELAY union bces_tc_ABORT )
     = empty &
6 bces_tc_ABORT inter (bces_tc_DELAY union bces_tc_DOUBLE_DELAY )
     = empty &
```

Figure 7.27.: Translation rule STRM for the translation of constraints on a set of variables representing a stream $strmName$ with possible messages $m \in Univ \cup \{\varepsilon\}$ to a Mona expression.

ponents and instantiates existing predicates of the subcomponent's component types according to the definition of the composed component. The translation rule instantiates the predicates with their input streams and output streams as parameters.

The input streams for the subcomponents are named after the source port of the incoming connector. It is important to select the corresponding stream not only by the target port *con.tgtPort* but also by the name of the target subcomponent *con.tgtCmp*. In case that the composed component has multiple subcomponents of the same component type a match by the port is not unique. The corresponding Mona variables are created by rule C1 shown in Figure 7.25 for parent-to-child connectors and by rule C2 shown in Figure 7.26 for child-to-child connectors. The input streams on input ports of subcomponents without an incoming connector are set to the value ε^{∞}, which is expressed in Mona by empty sets (`empty`).

The last quantification in the translation rule C4 shown in Figure 7.29 lists the Mona variables for the output streams of the instantiated subcomponent. These variables are

Translation Rule

Translation rule:

C3 `# parent-to-parent connectors`

$$\forall con \in \{con \in CCons \mid con.srcCmp = cType = con.tgtCmp\} :$$
$$\forall val \in con.tgtPort.type :$$
$$cType\_con.srcPort\_val \;=\; cType\_con.tgtPort\_val \;\&$$

Result of application to composed component type definition `DifferentDelays` from Figure 7.68:

Mona

```
1 DifferentDelays_i1_false = DifferentDelays_o1_false &
2 DifferentDelays_i1_true = DifferentDelays_o1_true &
```

Figure 7.28.: Translation rule C3 for the translation of parent-to-parent connectors into Mona statements.

named after the name of the subcomponent and the output port. The variables are declared by rule C1 presented in Figure 7.25 for child-to-parent connectors and by rule C2 shown in Figure 7.26 for child-to-child connectors.

Translation Rule

Translation rule:

C4 { *iteration over all subcomponents* $sub \in CSubCmps$
`# subcomponent` *sub.name* `of type` *sub.type*
sub.type.cType `(`
$\forall p \in sub.type.CPorts_{IN}$: *subcomponent input*
if $(\nexists con \in CCons : con.tgtCmp = sub.name \wedge con.tgtPort = p)$ **then**
$\forall val \in p.type$:
`empty,` *input port not connected: input is* ε^{∞}
else *input port is connected*
$\forall val \in p.type$: **let** $con = THE\ con \in CCons :$
$con.tgtCmp = sub.name \wedge con.tgtPort = p$ **in**
con.srcCmp`_`*con.srcPort.name*`_`*val* `,`
$\forall p \in sub.type.CPorts_{OUT}$: *subcomponent output from this port*
$\forall val \in p.type$:
sub.type.cType`_`*p.name*`_`*val* `,`
`# add common time to subcomponents`
`allTime)`
}$^{\&}_{sub \in CSubCmps}$ `;`

Result of application to composed component type definition `BumperBotES-Controller` from Figure 7.23:

Mona

```
1 # subcomponent bces of type BumpControlES
2 BumpControlES(BumperBotESController_emgStp_false,
     BumperBotESController_emgStp_true,
     BumperBotESController_bump_false,
     BumperBotESController_bump_true, timer_ts_ALERT,
     bces_rMot_FORWARD, bces_rMot_BACKWARD, bces_rMot_STOP,
     bces_lMot_FORWARD, bces_lMot_BACKWARD, bces_lMot_STOP,
     bces_tc_DELAY, bces_tc_DOUBLE_DELAY, bces_tc_ABORT, allTime)
3 &
4 # subcomponent timer of type Timer
5 Timer(bces_tc_DELAY, bces_tc_DOUBLE_DELAY, bces_tc_ABORT,
     timer_ts_ALERT, allTime);
```

Figure 7.29.: Translation rule C4 for the translation of subcomponents of composed component type definitions into Mona predicate instantiations.

7.3.4. Translation of ${}^*\text{MAA}_{ts}$ into Mona

We have presented two definitions for time-synchronous MontiArcAutomaton automata in Section 6.4: MAA_{ts} automata as the basic structure used for the definition of the automata's semantics and ${}^*\text{MAA}_{ts}$ automata whose structure is closer to the abstract syntax of MontiArcAutomaton models. We can translate ${}^*\text{MAA}_{ts}$ automata into MAA_{ts} automata using the reference removal from Definition 6.22, enabledness expansion from Definition 6.24, and different completions of the transition system to remove syntactic underspecification based on the symbol $*$. For finite input and output types enabledness expansion and chaos completion may increase the number of tuples in the transition relation δ of a ${}^*\text{MAA}_{ts}$ automaton to $(|S|+1)^2 * (\Pi_{var \in cmp.CVars} |\vec{V}_{var}|)^2 * \Pi_{p \in cmp.CPorts_{IN}} |\vec{I}_p| * \Pi_{p \in cmp.CPorts_{OUT}} |\vec{O}_p|$.

One advantage of our translation into Mona is that we avoid the expansion of the abstract syntax (transition relation) of the input models and handle completions inside the model checker Mona in a more compact, declarative form. The input of our translation is a ${}^*\text{MAA}_{ts}$ automaton from Definition 6.20 and its component definition $cmp \in CTDefs$ from Definition 6.8.

We now present the translation rules for the translation of ${}^*\text{MAA}_{ts}$ automata, that are used as implementations, into Mona. This translation contains the reference removal from Definition 6.22, the enabledness expansion as defined in Definition 6.24, and the ε_* completion to the transition system as defined in Definition 6.25. We show how to adapt the rules for chaos completion and I/O completion, when using ${}^*\text{MAA}_{ts}$ automata as specifications, in Section 6.4.4.

We use the automaton inside component `ToggleSwitch` as shown in Listing 7.30 to illustrate the translation rules. The complete translation result is included in Appendix L.2.

An overview of the translation rules for ${}^*\text{MAA}_{ts}$ automata to the concrete syntax of Mona is given in Figure 7.31. The rule A1 translates the input and output ports of the component definition into the head of a Mona predicate. The rules A2 and A3 translate the states and variables of the automaton into Mona. The rules A4 and A5 translate the initial states, their outputs, and the initial values of variables. The rules A3 and A5 are only executed if the component *cmp* has local variables ($cmp.CVars \neq \emptyset$). Finally, the translation rule A6 translates the transition system of the automaton.

The translation rule A1 shown in Figure 7.32 defines the head of the Mona predicate which describes the behavior of the MontiArcAutomaton automaton. The first part of rule A1 iterates over the input ports and defines one second order parameter for each possible value per input stream (see the representation of streams in Mona in Section 7.3.2). The stream on input port `pressed` of type `Boolean` (Listing 7.30, line 6) is translated into the two variables `pressed_false` and `pressed_true` as shown in the lines 3 and 4 of the listing in the lower part of Figure 7.32. The second part of the rule A1 (second universal quantification) does the same for all output ports. The third part adds the parameter `allTime` to the predicate. For component composition and behavior analysis, this parameter enables the synchronization of all components on one common finite representation of time. The result of the application of the rule A1 to the

MontiArcAutomaton

```
package example;

component ToggleSwitch {

  port
    in Boolean pressed,
    out Boolean active;

  automaton {
    state off, on;

    initial off / {active = false};

    off -> off {pressed = false} / {active = false};
    off -> on  {pressed = true}  / {active = true};
    on  -> on  {pressed = false} / {active = true};
    on  -> off {pressed = true}  / {active = false};
  }
}
```

Listing 7.30: A model of the component `ToggleSwitch` given in MontiArcAutomaton syntax.

component `ToggleSwitch` from Listing 7.30 is shown in the lower part of Figure 7.32.

The translation rule A2 shown in Figure 7.33 defines the automaton's states as existentially quantified second order variables. The representation is very similar to that of streams in Mona. A difference is that the sequence of states may not contain the value ε_ι. The automaton is in a state $s \in S$ at time $t \in \mathbb{N}$ if and only if the Mona set `s` contains the value t.

The first condition after the existential quantification is that, at any point in time the automaton is at least in one of its defined states. The last part of rule A2 additionally states that the automaton is in at most one of its states at any point in time.

The existential quantification over the state sequence requires a sequence of well-defined states that satisfies the further constraints imposed by the automaton in rules A3 to A6. The result of the application of the rule A2 to the component `ToggleSwitch` from Listing 7.30 with the two states `on` and `off` is shown in the lower part of Figure 7.33.

Rule A3 is shown in Figure 7.34 and translates the local variables of the automaton similar to the translation of the automaton's states. Again, the existence of a single value for each variable at any point in time is stated. In fact, our translation handles the states and local variables of the automaton in the same way.

The translation rule A4 shown in Figure 7.35 handles the translation of initial states and their initial outputs. The outer iteration creates a disjunction of the statements inside the iteration over the initial states and outputs $(s, \vec{o}) \in \iota$. The first line inside the

Translation Rule

Overview of the translation of a $^*\text{MAA}_{ts}$ automaton into a Mona predicate:

```
# predicate for component cmp.cType
executeRule(A1) ℹ see Figure 7.32 for body of rule A1
# states of the automaton
executeRule(A2) ℹ see Figure 7.33 for body of rule A2
if (cmp.CVars ≠ ∅) then
   # local variables of automaton
   executeRule(A3) ℹ see Figure 7.34 for body of rule A3
# assignment of initial state and output at time 0
executeRule(A4) ℹ see Figure 7.35 for body of rule A4
if (cmp.CVars ≠ ∅) then
   # assignment of values of local variables at time 0
   executeRule(A5) ℹ see Figure 7.37 for body of rule A5
# transitions restricted to valid times from allTime
executeRule(A6) ℹ see Figure 7.38 for body of rule A6
```

Figure 7.31.: Overview of the translation of $^*\text{MAA}_{ts}$ automata into the concrete syntax of the Mona language. The comments above the rule execution commands belong to the respective rules and are reproduced in this listing to give an intuition of the rules.

iteration states that the automaton is in the state s from the pair $(s, \vec{o})$ at time 0. The second line lists the outputs on all ports $p \in cmp.CPorts_{OUT}$ as defined in the second element $\vec{o}$ of the pair $(s, \vec{o})$. The value of the stream on the port at time 0 is assigned using the parametrized translation rule VAL shown in Figure 7.36. A special translation is necessary since the initial value $\vec{o}_p$ on an output port $p \in cmp.CPorts_{OUT}$ may be set to the special marker $*$. The translation rule VAL performs the ε completion for the initial outputs (see Definition 6.25, Item 1), i.e., in case of $\vec{o}_p = *$ the rule VAL states that there is no message on the stream at time 0, which is the implementation of the symbol ε in streams in Mona (see Section 7.3.2).

The translation rule A5 shown in Figure 7.37 translates the initial variable assignment γ into Mona as a constraint that needs to be satisfied. The translation rule adds a conjunct for every variable $var \in cmp.CVars$ and its value defined by γ_{var} fixing the value of the variable at time 0.

The result of an application of rule A5 to a $^*\text{MAA}_{ts}$ with a component definition containing the single variable declaration `Boolean open = false;` is shown in the listing in the lower part of Figure 7.37.

Translation Rule

Translation rule:

A1 pred *cmp.cType* (
input streams and their values
$\forall p \in cmp.CPorts_{IN}\ \forall v \in \vec{I}_p$:
var2 *p.name*_*v*,
output streams and their values
$\forall p \in cmp.CPorts_{OUT}\ \forall v \in \vec{O}_p$:
var2 *p.name*_*v*,
common time range allTime
var2 allTime) =

Result of application to automaton of component `ToggleSwitch` from Listing 7.30:

Mona

```
1 pred ToggleSwitch (
2 # input streams and their values
3   var2 pressed_true,
4   var2 pressed_false,
5 # output streams and their values
6   var2 active_true,
7   var2 active_false,
8 # common time range allTime
9   var2 allTime ) =
```

Figure 7.32.: Translation rule A1 for representing the head of the predicate of $^*\text{MAA}_{ts}$ automata in Mona.

The translation rule A6 shown in Figure 7.38 demonstrates the translation of the transition system of the automaton including its completion. All previous rules have introduced the necessary structures for input streams, states, variables and output streams as well as the initial values at time `0`. The transition system of the automaton determines the state, output, and variable assignments for time `t+1` based on the state, input, and variable values at time `t`.

Because all second order variables in Mona are finite sets, a Mona formula is unsatisfiable if it contains a constraint requiring a set to be infinite. One such constraint would be that for every state at time `t` we have a successor state at time `t+1` (the sequence of states represented as sets would need to be infinite). To prevent this cause of non-satisfaction, the translation rule A6 in Figure 7.38 adds an implication to the predicate guarding the quantification over times `t` by requiring that the time `t+1` is still a valid

Translation Rule

Translation rule:

A2 # states of the automaton
ex2 $\{\!|s|\!\}_{s\in S}^{'}$:
at least one state active at any point in time
$\{\!|s|\!\}_{s\in S}^{\text{union}}$ = allTime &
at most one state active at any point in time
$\forall s_1 \in S$:
s_1 inter ($\{\!|s_2|\!\}_{s_2\neq s_1\in S}^{\text{union}}$) = empty &

Result of application to automaton of component ToggleSwitch from Listing 7.30:

Mona

```
1 # states of the automaton
2 ex2 on, off:
3 # at least one state active at any point in time
4   on union off = allTime &
5 # at most one state active at any point in time
6   on inter (off ) = empty &
7   off inter (on ) = empty &
```

Figure 7.33.: Translation rule A2 for the translation of the states of *MAA$_{ts}$ automata into Mona.

point in time defined in the set allTime. Thus, all transitions from time t to time t+1 with t+1 in allTime satisfy the constraints introduced by the translation rules A7 and A8 executed inside the translation rule A6. For all other times the behavior of the component in Mona is not specified.

When using the *MAA$_{ts}$ automata translations for the verification of properties we always assert that the set allTime contains the time 0 and all intermediate times up to its latest point in time. We check that the analyzed property holds for all finite sets allTime (see Section 7.3.2, Listing 7.20). This *external* quantification over allTime ensures that the analysis result is valid for all finite input and output streams and prevents contradictions based on infinite sets inside the predicates of the components.

The translation rule A7 shown in Figure 7.39 translates the transitions relation δ of a *MAA$_{ts}$ automaton into a disjunction over the possible transitions of the automaton. Each tuple $(s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta$ results in a conjunction of constraints over the source state, the inputs, and the variable values at time t and the target state, the outputs, and the variable assignments at time t+1. To satisfy the disjunction at least

Translation Rule

Translation rule:

A3 # local variables of automaton

ex2 $\{\!|var.name \_ v|\!\}^{\vdash}_{var \in cmp.CVars,\ v \in \vec{V}_{var}}$:

$\forall var \in cmp.CVars$:

at least one value at any point in time

$\{\!|var.name \_ v|\!\}^{\text{union}}_{v \in \vec{V}_{var}}$ = allTime &

at most one value at any point in time

$\forall v_1 \in \vec{V}_{var}$:

$var.name \_ v_1$ inter ($\{\!|var.name \_ v_2|\!\}^{\text{union}}_{v_1 \neq v_2 \in \vec{V}_{var}}$) = empty &

Result of application to a component *cmp* with $cmp.CVars = \{(\texttt{open}, \texttt{Boolean}\}$, e.g., the variable declaration `Boolean open = false;`:

Mona

```
1 ex2 open_false, open_true:
2   open_false union open_true = allTime &
3   open_false inter (open_true ) = empty &
4   open_true inter (open_false ) = empty &
```

Figure 7.34.: Translation rule A3 for the variables of $^*\text{MAA}_{ts}$ automata to Mona.

one constraint on the states, variables, input, and output imposed by one transition needs to be satisfied.

The translation of the transitions realizes the enabledness expansion for $^*\text{MAA}_{ts}$ automata of Definition 6.24 (without supporting guards). The expansion from Definition 6.24 adds tuples to the transition relation in case an input port or a variable value is marked with the special symbol $*$. The additional tuples match all values of the port or variable that was marked with the symbol $*$. The translation rule implements the expansion by not adding a constraint for values marked with the symbol $*$. The checks $\vec{i}_p \neq *$ and $\vec{v}_{var} \neq *$ are shown in the first two quantifications inside the translation rule A7 in Figure 7.39. Thus, one tuple $(s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta$ from a $^*\text{MAA}_{ts}$ automaton is expressed as a single disjunction in Mona that might represent multiple tuples of the enabledness expanded transition relation.

The translation rule A7 also handles the reference removal defined in Definition 6.22 that removes references from variables, input ports, and output ports to other elements. The reference removal is explicitly handled inside rule VAL shown in Figure 7.36. The second condition checked in rule VAL is whether the message in its parameter m refers to a local variable or a port. In this case the rule generates a constraint about the equality of the referencing and the referenced elements values at time `t`.

Translation Rule

Translation rule:

A4 `# assignment of initial state and output at time 0`

(⦃ `(0 in` s *quantification over initial states and output* $(s, \vec{o}) \in \iota$

$\forall p \in cmp.CPorts_{OUT}$:

& **executeRule(**VAL `0`, $p.name$, $\vec{o}_p$, $\vec{O}_p$**)**

) ⦄$_{(s,\vec{o})\in\iota}$) &

Result of application to automaton of component `ToggleSwitch` from Listing 7.30:

Mona

```
1 # assignment of initial state and output at time 0
2 ((0 in off & 0 in active_false )) &
```

Figure 7.35.: Translation rule A4 for the initial states and output of $^*\text{MAA}_{ts}$ automata in Mona.

Translation Rule

Translation rule with parameters t, $prefix$, m, $Univ$, and optional t_{src}:

VAL **if (**$m = \varepsilon \vee m = *$**) then** *assign no value*

t `notin` ⦃$prefix\_v$⦄$^{\text{union}}_{v\in Univ}$

else if ($m \in cmp.CVars \cup cmp.CPorts$**) then**

assign value of referenced variable/port at time t_{src}

⦃t `in` $prefix\_val$ `<=>` t_{src} `in` $m.name\_val$⦄$^{\&}_{val\in Univ}$

else

t `in` $prefix\_m$

Result of application to parameters t = `0`, $prefix$ = `pressed`, $m = \varepsilon$, and $Univ$ = {**true**, **false**}:

Mona

```
1 0 notin (pressed_false union pressed_true)
```

Figure 7.36.: Translation rule VAL for the translation of a message $m \in Univ \vee m = *$ on a port $prefix$ at time t into a Mona expression.

Translation Rule

Translation rule:

A5 `# assignment of values of local variables at time 0`

$$\{\texttt{0 in } var.name \leftrightarrow \gamma_{var}\}_{var \in cmp.CVars}^{\&}$$

Result of application to a component cmp with $cmp.CVars = \{(\texttt{open}, \texttt{Boolean}\}$ and $\gamma_{\texttt{open}} =$ **false**, i.e., the variable declaration `Boolean open = false;`:

Mona

```
1 # assignment of values of local variables at time 0
2 0 in open_false &
```

Figure 7.37.: Translation rule A5 for the initial values of variables of ${}^*\text{MAA}_{ts}$ automata in Mona.

Translation Rule

Translation rule:

A6
```
# transitions restricted to valid times from allTime
all1 t: t+1 in allTime => (
   # transitions of the automaton
   ( executeRule(A7) )  see Figure 7.39 for body of rule A7
   # completion of the transition system
   | executeRule(A8)  see Figure 7.41 for body of rule A8
);
```

Figure 7.38.: Translation rule A6 for the transition system of ${}^*\text{MAA}_{ts}$ automata and its ε_* completion to Mona.

The translation of the transitions in rule A7 from Figure 7.39 also implements Item 2 of Definition 6.25 of the ε_* completion. The third quantification in the translation rule A7 shown in Figure 7.39 uses the translation rule VAL from Figure 7.36, which translates the symbol $*$ on an output port $p \in cmp.CPorts_{OUT}$ into not sending any message on the corresponding stream at time `t+1`. Assignments of values other than $*$ to the variables at time `t+1` are translated in the fourth quantification in rule A7. Variable assignments that are marked with the symbol $*$ are translated in the last quantification inside rule A7 into instantiations of the predicate `sameNextValue` shown in Listing 7.40. The predicate states that the value of a variable at time `t` is equal to its value at time `t+1`.

A single variable of an automaton is represented by one set for each possible value. Due to our representation of variables, the rule adds one instantiation of the predicate `sameNextValue` for every set used for the encoding of the variable.

Translation Rule

Translation rule:

A7 `# transitions of the automaton`

$\{$ `( t in` s_{src} *source state, inputs, and variable values at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

`&` **executeRule(**VAL `t`, $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

`&` **executeRule(**VAL `t`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

`& t+1 in` s_{tgt} *target state, outputs, and variable values at time t+1*

$\forall p \in cmp.CPorts_{OUT}$:

`&` **executeRule(**VAL `t+1`, $p.name$, $\vec{o}_p$, $\vec{O}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{a}_{var} \neq *\}$

`&` **executeRule(**VAL `t+1`, $var.name$, $\vec{a}_{var}$, $\vec{V}_{var}$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{a}_{var} = *\}$

`&` $\{$`sameNextValue(`$var.name$`_`val`, t)`$\}^{\&}_{val \in \vec{V}_{var}}$

`)` $\}^{|}_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a}) \in \delta}$

Result of application to automaton of component `ToggleSwitch` from Listing 7.30:

Mona

```
1 # the transitions of the automaton
2 ( t in off  & t in pressed_false
3         & t+1 in off & t+1 in active_false ) |
4 ( t in off  & t in pressed_true
5         & t+1 in on  & t+1 in active_true  ) |
6 ( t in on   & t in pressed_false
7         & t+1 in on  & t+1 in active_true ) |
8 ( t in on   & t in pressed_true
9         & t+1 in off & t+1 in active_false )
```

Figure 7.39.: Translation rule A7 for the transitions of $^*\text{MAA}_{ts}$ automata to Mona.

Finally, the translation rule A8 is shown in Figure 7.41. This rule expresses the completion of the transition system for $\varepsilon_{\circlearrowleft}$ completion from Definition 6.25, Item 3 in Mona. The last step of $\varepsilon_{\circlearrowleft}$ completion adds those transitions where no transition is defined in the transition system δ' for a given source state $s_{src} \in S$, a given input $\vec{i} \in \vec{I}$,

Mona

```
# the value at time t is also the value at time t+1
pred sameNextValue(var2 value, var1 t) =
  t in value <=> t+1 in value;
```

Listing 7.40: Predicate for sets representing values on streams to state that the value does not change from time t to time $t+1$.

and the variable values $\vec{v} \in \vec{V}$. The added transitions define the target state $s_{tgt} = s_{src}$, the output $\vec{\varepsilon}$ (interpreted as not sending a message), and a variable assignment of the previous values:

$$\{(s_{src}, \vec{i}, \vec{v}, s_{src}, \vec{\varepsilon}, \vec{v}) \mid \nexists s'_{tgt}, \vec{o}', \vec{a}' .\ (s_{src}, \vec{i}, \vec{v}, s'_{tgt}, \vec{o}', \vec{a}') \in \delta'\}$$

The first part of the translation rule A8 shown in Figure 7.41 is a negation of the enabledness conditions of all transitions of the automaton. The enabledness conditions are based on the source state, the inputs on all ports, and the values of all variables at time `t`. For every transition defined in δ these conditions are identical with the first part of the constraint created by the translation rule A7 (only covering s_{src}, $\vec{i}$, and $\vec{v}$).

The negation of the enabledness conditions expresses that none of the defined transitions is enabled. The translation rule adds conjuncts that the state at time `t+1` is the same as the state at time `t`. Again, this is done by instantiating the predicate `sameNextValue` from Listing 7.40 for all the variables representing the state of the automaton. Next, the translation rule adds one conjunct for every output port stating that no message is sent at time `t+1` by executing the rule VAL (see Figure 7.36) with the output port and the value ε as parameters. The translation rule A8 finally adds a conjunct for every local variable to ensure that its value does not change from time `t` to time `t+1`.

The formula generated by the translation rule A7 for the transitions defined by the automaton and the formula generated by the translation rule A8 for previously undefined behavior are combined into one disjunction in the translation rule A6 shown in Figure 7.38. This ensures that either one of the given transitions is taken or that the current state and variable values are preserved and no message is sent.

Translation Rule

Translation rule:

A8 `# ε completion of the transition system`

$\sim$ ({ (`t in` s_{src} *no transition enabled at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

& **executeRule**(VAL `t`, $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t`)

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

& **executeRule**(VAL `t`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`)

)

$\bigvee_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a})\in\delta}$

)

$\forall s \in S$: *stay in same state*

& `sameNextValue(`s`, t)`

$\forall p \in cmp.CPorts_{OUT}$: *send* ε *on all ports*

& **executeRule**(VAL `t+1`, $p.name$, ε, $\vec{O}_p$)

$\forall var \in cmp.CVars$: *preserve values of variables*

& { `sameNextValue(`$var.name$`_`val`, t)` }$^{\&}_{val\in\vec{V}_{var}}$

Result of application to automaton of component `ToggleSwitch` from Listing 7.30:

Mona

```
1 # completion of the transition system
2 ~ ( ( t in off  & t in pressed_false) |
3     ( t in off  & t in pressed_true) |
4     ( t in on   & t in pressed_false) |
5     ( t in on   & t in pressed_true) )
6 & sameNextValue(on, t)
7 & sameNextValue(off, t)
8 & t+1 notin (active_false union active_true)
```

Figure 7.41.: Translation rule A8 for the ε completion of the transition system of $^*\text{MAA}_{ts}$ automata in Mona. In case no transition is enabled the component does not send any message on any port (ε) and preserves the values of the variables.

Chaos completion translation rules

Chaos completion of the behavior of a *MAA$_{ts}$ automaton is applied when using the automaton as an incomplete specification. Chaos completion completes the inputs, outputs, variable values and assignments, and target states with all possible combinations in case these are not specified by the automaton.

Technically, chaos completion adds a special state `Chaos` that the automaton may enter whenever its reaction to a state, input, and variable combination is not defined by a transition. The behavior in the state `Chaos` is arbitrary. Chaos completion for enabledness expanded *MAA$_{ts}$ automata is formally defined in Definition 6.28. An example specification of the behavior of the component `ToggleSwitch` is shown in Listing 7.42. We have used this example in Section 6.4.4 to illustrate chaos completion. A graphical representation of the component and automaton is shown in Figure 6.27. The chaos completion of the transition relation δ of the automaton is illustrated in Figure 6.29.

MontiArcAutomaton

```
1  package example;
2
3  component ToggleSwitchSpec {
4
5    port
6      in Boolean pressed,
7      out Boolean active;
8
9    automaton {
10     state s1;
11
12     initial / {active = false};
13
14     s1 -> s1 {pressed = false} / {active = false};
15   }
16 }
```

Listing 7.42: A specification for the behavior of the component `ToggleSwitch`. The component `ToggleSwitchSpec` is shown in a graphical representation in Figure 6.27.

The difference between ε_* completion and chaos completion requires updating the previously presented translation rules to support chaos completion. For chaos completion we replace the translation rule A2 (Figure 7.33) by the translation rule A2$_c$ (Figure 7.43), the translation rule A4 (Figure 7.35) by the translation rule A4$_c$ (Figure 7.44), the translation rule A7 (Figure 7.39) by the translation rule A7$_c$ (Figure 7.45), and the translation rule A8 (Figure 7.41) by the translation rule A8$_c$ (Figure 7.46).

The translation rule $A2_c$ shown in Figure 7.43 implements Item 1 of Definition 6.28. It adds the state `Chaos` to the set of states in the Mona translation of the automaton.

Translation Rule

Translation rule:

$A2_c$ `# states of the automaton`
`ex2` $\{\!|s|\!\}_{s\in S}^{,}$`, Chaos:`
`# at least one state active at any point in time`
$\{\!|s|\!\}_{s\in S}^{\text{union}}$ `union Chaos = allTime &`
`# at most one state active at any point in time`
$\forall s_1 \in S \cup \{\texttt{Chaos}\}$:
s_1 `inter (` $\{\!|s_2|\!\}_{s_2\neq s_1\in S\cup\{\texttt{Chaos}\}}^{\text{union}}$ `) = empty &`

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # states of the automaton
2 ex2 s1, Chaos:
3 # at least one state active at any point in time
4   s1 union Chaos = allTime &
5 # at most one state active at any point in time
6   s1 inter (Chaos ) = empty &
7   Chaos inter (s1 ) = empty &
```

Figure 7.43.: Translation rule $A2_c$ for the translation of the states of $^*MAA_{ts}$ automata into Mona.

The translation rule A4*c* is shown in Figure 7.44 and implements Item 2 of Definition 6.28. The only difference between the rule $A4_c$ and the rule A4 shown in Figure 7.35 is in the line with the universal quantification over the output ports. In case the initial output on port $p \in cmp.CPorts_{OUT}$ is unspecified $\vec{o}_p = *$ the rule $A4_c$ does not add a constraint on the sets representing the output at time `0`.

Translation Rule

Translation rule:

$A4_c$ `# assignment of initial state and output at time 0`

$$(\ \{\!|\ (0 \text{ in } s$$
$$\forall p \in \{p \in cmp.CPorts_{OUT} \mid \vec{o}_p \neq *\} :$$
$$\&\ \textbf{executeRule(}\text{VAL } 0, p.name, \vec{o}_p, \vec{O}_p\textbf{)}$$
$$)\ |\!\}_{(s,\vec{o})\in\iota}\ )\ \&$$

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # assignment of initial state and output at time 0
2 ((0 in s1 & 0 in active_false )) &
```

Figure 7.44.: Translation rule $A4_c$ for the initial states and output of ${}^*MAA_{ts}$ automata in Mona.

The translation rule A7$_c$ is shown in Figure 7.45 and implements Item 3 of Definition 6.28. One difference between the rule A7$_c$ and the rule A7 shown in Figure 7.39 is that rule A7$_c$ does not add any constraint to the generated formula if the output on a port $p \in cmp.CPorts_{OUT}$ is undefined ($\vec{o}_p = *$). Another difference is that the variable value of a variable $var \in cmp.CVars$ is not preserved in case the assignment is undefined ($\vec{a}_{var} = *$).

Translation Rule

Translation rule:

A7$_c$ `# transitions of the automaton`

`{ ( t in` s_{src} *source state, inputs, and variable values at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

`&` **executeRule(**VAL `t`, $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

`&` **executeRule(**VAL `t`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

`& t+1 in` s_{tgt} *target state, outputs, and variable values at time t+1*

$\forall p \in \{p \in cmp.CPorts_{OUT} \mid \vec{o}_p \neq *\}$:

`&` **executeRule(**VAL `t+1`, $p.name$, $\vec{o}_p$, $\vec{O}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{a}_{var} \neq *\}$

`&` **executeRule(**VAL `t+1`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

`)` `}` $_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a})\in\delta}$

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # the transitions of the automaton
2 ( t in s1  & t in pressed_false
3         & t+1 in s1 & t+1 in active_false )
```

Figure 7.45.: Translation rule A7$_c$ for the transitions of *MAA$_{ts}$ automata to Mona.

The translation rule A8$_c$ is shown in Figure 7.46 and implements Item 4 of Definition 6.28. The difference between the rule A8$_c$ and the rule A8 shown in Figure 7.41 is that rule A8$_c$ does not add any constraint to the generated formula in case none of the defined transitions is enabled. The translation rule A6 from Figure 7.38 creates a disjunction of the formulas created by the translation rule A7$_c$ and the rule A8$_c$. Thus, either one of the transitions defined in the *MAA$_{ts}$ automaton is executed at time `t`, i.e., the target state, the output, and the variable assignment for time `t+1` are determined, or none of the transitions was enabled at time `t` and the target state, the output, and the variable assignments for time `t+1` are not further constrained. This corresponds to a completion with all possible transitions as defined in Definition 6.28, Item 4.

Translation Rule

Translation rule:

A8$_c$ `# chaos completion of the transition system`

`~ ( {| ( t in` s_{src} *no transition enabled at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

`&` **executeRule(**VAL `t`, $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

`&` **executeRule(**VAL `t`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

`) |}` $_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a})\in\delta}$ `)`

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # chaos completion of the transition system
2 ~ ( ( t in s1 & t in pressed_false ) )
```

Figure 7.46.: Translation rule A8$_c$ for the chaos completion of the transition system of *MAA$_{ts}$ automata in Mona. In case no transition is enabled the behavior is not constrained.

Output completion translation rules

Output completion is less permissive than chaos completion. It may be used to specify complex behavior using multiple states with unknown output behavior for some or all of the states and transitions. The symbol $*$ as the output on a port is interpreted as all possible values. If the automaton receives an input for which no transition of the automaton is enabled it produces an arbitrary output. However, it stays in the state that it is in and preserves the values of the local variables. Output completion is formally defined in Definition 6.30.

The translation of a ${}^*\mathrm{MAA}_{ts}$ automaton into Mona using output completion requires the adaption of some of the translation rules A1 to A8 for the translation of automata. For output completion we replace the translation rule A4 (Figure 7.35) by the translation rule $\mathrm{A4}_c$ (Figure 7.44), the translation rule A7 (Figure 7.39) by the translation rule $\mathrm{A7}_r$ (Figure 7.47), and the translation rule A8 (Figure 7.41) by the translation rule $\mathrm{A8}_r$ (Figure 7.48).

Our implementation of output completion through the translation rules replaces the rule A4 of the translation implementing ε_* completion with the translation rule $\mathrm{A4}_c$ presented in Figure 7.44 that we have already used for the implementation of chaos completion. The execution of the rule $\mathrm{A4}_c$ results in a formula that does not constrain the initial output on a port if it is marked with $*$ in the ${}^*\mathrm{MAA}_{ts}$ automaton. This rule implements Item 1 of Definition 6.30.

The rule $A7_r$ shown in Figure 7.47 implements Item 2 of Definition 6.30. The only difference to rule A7 shown in Figure 7.39 is the third universal quantification in the middle of the rule, which is now restricted to defined outputs ($\vec{o}_p \neq *$). For outputs on ports set to $*$ no constraint is added to the formula generated by this translation rule.

A difference of this rule compared to the translation rule $A7_c$ from Figure 7.45 for chaos completion is that variable values are preserved if their assignment is set to $*$ (see last quantification in rule $A7_r$ in Figure 7.47).

Translation Rule

Translation rule:

$A7_r$

`# transitions of the automaton`

`{ ( t in` s_{src} *source state, inputs, and variable values at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

`&` **executeRule(**VAL `t`, $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

`&` **executeRule(**VAL `t`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

`& t+1 in` s_{tgt} *target state, outputs, and variable values at time t+1*

$\forall p \in \{p \in cmp.CPorts_{OUT} \mid \vec{o}_p \neq *\}$:

`&` **executeRule(**VAL `t+1`, $p.name$, $\vec{o}_p$, $\vec{O}_p$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{a}_{var} \neq *\}$

`&` **executeRule(**VAL `t+1`, $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t`**)**

$\forall var \in \{var \in cmp.CVars \mid \vec{a}_{var} = *\}$

`&` `{sameNextValue(`$var.name$`_`val`, t)}`$^{\&}_{val \in \vec{V}_{var}}$

`)` $\Big\}^{|}_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a}) \in \delta}$

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # the transitions of the automaton
2 ( t in s1  & t in pressed_false
3         & t+1 in s1 & t+1 in active_false )
```

Figure 7.47.: Translation rule $A7_r$ for the transitions of ${}^*MAA_{ts}$ automata into Mona.

The rule A8$_r$ shown in Figure 7.48 implements Item 3 of Definition 6.30. The difference to rule A8 from Figure 7.41 is that the lower part of the rule does not add any constraints on possible outputs at time `t+1`. The difference of rule A8$_r$ to the rule A8$_c$ shown in Figure 7.46 is that the two quantifications at the bottom or rule A8$_r$ add constraints on preserving the state and variable values of the automaton.

Translation Rule

Translation rule:

A8$_r$ `# output completion of the transition system`

`~ ( {| ( t in` s_{src} *no transition enabled at time t*

$\forall p \in \{p \in cmp.CPorts_{IN} \mid \vec{i}_p \neq *\}$:

`&` **`executeRule`**`(VAL t,` $p.name$, $\vec{i}_p$, $\vec{I}_p$, `t)`

$\forall var \in \{var \in cmp.CVars \mid \vec{v}_{var} \neq *\}$:

`&` **`executeRule`**`(VAL t,` $var.name$, $\vec{v}_{var}$, $\vec{V}_{var}$, `t)`

`)` $|\!|\!\vdash_{(s_{src},\phi,\vec{i},\vec{v},s_{tgt},\vec{o},\vec{a})\in\delta}$ `)`

$\forall s \in S$: *stay in same state*

`& sameNextValue(`s`, t)`

$\forall var \in cmp.CVars$: *preserve values of variables*

`&` `{|sameNextValue(`$var.name$`_`val`, t)|}`$^{\&}_{val\in\vec{V}_{var}}$

Result of application to automaton of component `ToggleSwitchSpec` from Listing 7.42:

Mona

```
1 # response completion of the transition system
2 ~ ( ( t in s1 & t in pressed_false ) )
3 & sameNextValue(s1, t)
```

Figure 7.48.: Translation rule A8$_r$ for the output completion of the transition system of *MAA$_{ts}$ automata in Mona. In case no transition is enabled the automaton stays in the current state and leaves variable values unchanged.

7.3.5. Mona Dependency Resolution and Manual Mona Implementations

We discuss two implementation specific solutions of the representation of MontiArcAutomaton components in Mona. First, we show how to support our compositional code generation with a suitable dependency resolution not directly available in Mona. Second, we show how to provide manual implementations for component types in case neither an automaton nor a decomposition to subcomponents is specified for a MontiArcAutomaton component.

Mona dependency resolution

One important language feature of the modeling language MontiArcAutomaton is black box component composition. Our Mona translation carries over this concept to the generated predicates: the predicate for a component is generated independently of its future use. The compositional generation however requires a resolution of the dependencies of generated Mona artifacts. We first discuss how dependencies are declared in MontiCore modeling languages and then present our solution for Mona.

Modeling languages developed using the DSL framework MontiCore distinguish between two coordinates of models. Models are organized in physical files (artifacts) and therefore have a name in the computer's file system that identifies them. More importantly, models are also organized in virtual, hierarchical packages and every model has a unique name in its package. Thus, a model is also identified by its virtual coordinates consisting of a package and model name. MontiCore allows to handle models by their virtual coordinates. This enables a uniform handling of models from different physical sources, e.g., multiple model folders or model libraries in archives.

The composition of components in MontiArcAutomaton is done on the level of model names that are visible for the importing model. MontiCore supports a mechanism to make imported models visible to importing models [Vö11]. The input language of the tool Mona does not provide a distinction between the physical and virtual coordinates of an artifact. Mona only offers a basic `include` directive to include verbatim code from another file. Multiple inclusions of the same artifact lead to inconsistent input since Mona includes files verbatim. In addition, all dependencies of a Mona file have to be included before the file itself.

In our implementation of the translation of MontiArcAutomaton models into Mona we rebuild the virtual MontiArcAutomaton coordinates of packages and model names as physical artifact coordinates. We generate one Mona artifact for every MontiArcAutomaton component type definition in a folder corresponding to the package name of the component. The Mona artifact contains the predicate expressing the semantics of the MontiArcAutomaton model. Since Mona only offers verbatim inclusion of code we name predicates fully qualified after their component type definition. As an example, the component type definition `Timer` in the package `util` is translated into the Mona predicate `util_Timer` in a file `Timer.mona` in a subfolder `util`.

Figure 7.49 shows Mona `include` dependencies derived from MontiArcAutomaton component dependencies of a refinement check between two composed components. The

nodes in Figure 7.49 are labeled with the names of imported components.

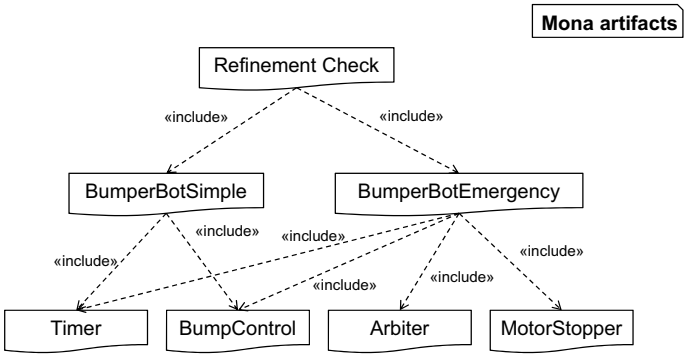


Figure 7.49.: Dependencies of Mona files of the MontiArcAutomaton models for the refinement check between the component `BumperBotSimple` and the component `BumperBotEmergency`.

We solve the problem of importing the relevant predicates for all Mona files by including the dependencies of each generated file as an annotation. To analyze the refinement check illustrated in Figure 7.49 we compute the order of the dependencies by a depth-first traversal of the directed acyclic dependency graph. During the traversal the algorithm lists the file names of the node for inclusion. The file names of all children are listed before the file names of their parents. The traversal descends to every node only once.

Consider the example of the Mona dependencies shown in Figure 7.49. In a dependency resolution stage of our execution of the Mona analysis the algorithm computes the list of `include` statements shown in Listing 7.50. This pre-processing allows the compositional code generation in separate Mona files independent of the way that the generated Mona predicates of the component types are used.

Mona

```
1 include "lib/Timer.mona";
2 include "bumperbot/BumpControl.mona";
3 include "bumperbot/BumperBotSimple.mona";
4 include "lib/Arbiter.mona";
5 include "lib/MotorStopper.mona";
6 include "bumperbot/BumperBotEmergency.mona";
```

Listing 7.50: The `include` statements in the parent Mona file generated from the dependency graph in Figure 7.49.

Artifact Kind	Project Folder
MontiArcAutomaton model	`/src/main/model/`
Mona code	`/src/main/mona/`
generated Mona code	`/target/generated-mona/`

Table 7.51.: Organization of artifacts in MontiArcAutomaton Mona project folders.

Generation gap for manual implementations

In some cases the behavior of a component can not be easily specified using an automaton or a composition of available components. To support manual implementations of MontiArcAutomaton components in Mona we have implemented a variant of the generation gap pattern [Vli98, Fow10] in our Mona code generator.

The generation gap pattern allows the separation of generated code from handwritten code. In case the generated code is regenerated the manual implementation is not lost as it might happen in an approach where generated code is modified. Implementations of the generation gap pattern are mostly seen in object-oriented programming where the generated code and the handwritten code are kept in multiple classes linked by inheritance [Fow10]. A very basic form of the pattern is the generation of an abstract class that is later extended by a manually written class.

A MontiArcAutomaton component type definition that does not contain an automaton and is not composed of other components is expected to come with a manual Mona implementation. To separate the manual implementations from generated code we organize Mona code, MontiArcAutomaton models, and generated code according to the structure shown in Table 7.51.

Our variant of the generation gap pattern generates a Mona file for the component that requires a manual implementation into the folder `/target/generated-mona/` to allow a uniform handling of generated artifacts independent of manual implementations. The generated Mona file declares a Mona predicate that points to an predicate with the suffix `Impl` specified in a file in the folder `/src/main/mona/`.

An example for a component with a manual implementation in Mona is the `Timer` component. The MontiArcAutomaton component type definition of the `Timer` component is shown in Listing 7.52. The code generator for the translation of MontiArcAutomaton components into Mona code detects that the component type definition contains no implementation — it is not decomposed and does not contain an automaton. The Mona predicate `lib_Timer` generated for the component `Timer` in the package `lib` is shown in Listing 7.53. The predicate depends on the manual implementation of the predicate `lib_TimerImpl` that is expected in the file `/src/main/mona/lib/-TimerImpl.mona`. The dependency is not declared in Mona using the `include` statement but in a comment that is processed in the dependency resolution stage of the Mona execution described before.

The generation gap pattern as implemented in our Mona code generation allows the regeneration of predicates for component type definitions without loosing manual Mona

MontiArcAutomaton

```
package lib;

component Timer {
    port
        in TimerCmd tc,
        out TimerSignal ts;
}
```

Listing 7.52: The MontiArcAutomaton component type definition of the component `Timer` in concrete syntax.

Mona

```
#DEPENDENCY "/src/main/mona/lib/TimerImpl.mona";

pred lib_Timer ( var2 Timer_tc_DELAY,
  var2 Timer_tc_DOUBLE_DELAY, var2 Timer_tc_ABORT,
  var2 Timer_ts_ALERT, var2 allTime )
  = lib_TimerImpl ( Timer_tc_DELAY, Timer_tc_DOUBLE_DELAY,
                    Timer_tc_ABORT, Timer_ts_ALERT, allTime );
```

Listing 7.53: The Mona predicate `lib_Timer` generated for the component type definition `Timer` from Listing 7.52.

implementations. However, changes to the interface of a component still require adaptions of the manually written Mona code.

7.4. Specifications and Specification Language

The translation introduced in Section 7.3 allows us to express the semantics of MontiArcAutomaton composed components and ${}^{*}\mathrm{MAA}_{ts}$ automata in WS1S logic. We may express both, behavior specifications and implementations, in the modeling language MontiArcAutomaton. With Mona we receive a framework for formal verification of the properties of these components.

We are mainly interested in checking whether one specification refines another to allow an iterative approach to systems development. Furthermore, we are interested in related questions, e.g., whether two specifications are equivalent with respect to their input and output or whether an implementation is a refinement of the conjunction of specifications. We now introduce a specification language for specifying these properties for MontiArcAutomaton specifications and implementations.

7.4.1. Specification Language

We have developed a MontiArcAutomaton specification language to assert relations between MontiArcAutomaton components that should be verified. Such a statement is called a check. Multiple checks are organized in a suite.

MontiArcAutomaton specification suite

The declaration of a MontiArcAutomaton specification suite starts with the keyword `suite`. It has a name and a body as shown in line 8 of Listing 7.54. Each suite is defined in its own file. The suite in Listing 7.54 is called `BumperBotRefinementSteps`. It imports three component type definitions `BumpControlSpec1a`, `BumpControlSpec1b`, and `BumpControlSpec1c` that specify the behavior of the component `BumpControl` (ll. 3-6). The specifications and the implementation of the component `BumpControl` were introduced as part of the example in Section 7.1.

MAASpecification

```
1  package bumperbot;
2
3  import bumperbot.BumpControlSpec1a;
4  import bumperbot.BumpControlSpec1b;
5  import bumperbot.BumpControlSpec1c;
6  import bumperbot.BumpControl;
7
8  suite BumperBotRefinementSteps {
9
10   check Spec1bRefinesSpec1a:
11     BumpControlSpec1b refines BumpControlSpec1a;
12
13   check ImplRefinesSpec1a1b:
14     BumpControl refines (BumpControlSpec1a and BumpControlSpec1b);
15
16   check NotImplEqualsSpec1c:
17     not BumpControl equals BumpControlSpec1c;
18 }
```

Listing 7.54: The MontiArcAutomaton specification suite `BumperBotRefinement-Steps` consisting of three MontiArcAutomaton specification checks.

MontiArcAutomaton specification check

A MontiArcAutomaton specification check is a statement about the relation of the behaviors of MontiArcAutomaton components. A check starts with the keyword `check` and a name followed by an assertion of the relation of the analyzed components. We distinguish between two relations:

refines the behavior of the component on the left refines the behavior of the component on the right on all shared ports

equals all input histories and their according output histories are valid I/O histories of either of the components if and only if they are accepted by the other component

Please note that the relation `refines` enables us to check both the refinement with possible upward simulation according to Definition 7.12 and the refinement for component replacement according to Definition 7.13. In both cases the syntactic requirements on the component interfaces need to be checked as defined in Definition 7.12 and Definition 7.13.

The check `Spec1bRefinesSpec1a` (see Listing 7.54, ll. 10-11) states that the behavior of the component `BumpControlSpec1b` (see Figure 7.5) refines the behavior of the component `BumpControlSpec1a` (see Figure 7.4).

It is possible to use multiple components on either side of the `refines` or `equals` statements. An example is shown in line 14 of Listing 7.54 in the check called `ImplRefinesSpec1a1b`. This check states that the component `BumpControl` shown in Figure 6.3 refines the conjunction of the specifications `BumpControlSpec1a` and `BumpControlSpec1b`. The conjunction expressed by the keyword `and` refers to the conjunction of the predicates over input and output streams, i.e., the implementation `BumpControl` refines both specifications.

We finally state that the implementation `BumpControl` does not equal the specification `BumpControlSpec1c` from Figure 7.6 (the most refined specification introduced in the example in Section 7.1). The keyword `not` (see Listing 7.54, l. 17) can be placed in front of every statement to specify that the relation `equals` or `refines` does not hold.

Example specification suites, also introduced in Section 7.6, are available from [wwws]. The complete grammar of the MontiArcAutomaton specifications suite language is shown in Appendix K. A definition of the structure of a MontiArcAutomaton specification check is shown in Definition 7.55. The sets *LeftSide* and *RightSide* contain the component type definitions on the left and right sides of the check. The well-formedness rule in Definition 7.55, Item 5 ensures that all shared ports, i.e., ports with the same name, have the same type.

Definition 7.55 (MontiArcAutomaton specification check)**.** A MontiArcAutomaton specification check is a structure $(negation, LeftSide, rel, RightSide)$ with

1. $negation \in \{\texttt{false}, \texttt{true}\}$ the possible negation of the check,
2. $LeftSide \subseteq CTDefs$ a non-empty set of component type definitions,
3. $rel \in \{\texttt{refines}, \texttt{equals}\}$ the relation to check, and
4. $RightSide \subseteq CTDefs$ a non-empty set of component type definitions.

With the well-formedness rule

5. $\forall p, p' \in \bigcup_{cmp \in (RightSide \cup LeftSide)} cmp.CPorts : p.name = p'.name \Rightarrow p.type = p'.type$, i.e., the port and type combinations of the checked component types are unique.

△

7.4.2. Checking Specifications and Witnesses for Non-Satisfaction

To verify a MontiArcAutomaton specification check statement we translate it into a verification problem for Mona and check it using the Mona tool [wwwz]. The translation of a MontiArcAutomaton specification check from Definition 7.55 is based on our translation of component types into Mona. The predicates generated from component type definitions are referenced by the formula checked by Mona.

We translate a specification check of the type `refines` into an implication between the conjunctions of the predicates for the components on the left side and the components on the right side of the check. A specification check of the type `equals` is translated into the equivalence of the conjunctions of the predicates for all inputs and outputs. The translation rule SC for the translation of a MontiArcAutomaton specification check is shown in Figure 7.56.

The first part of the translation rule SC presented in Figure 7.56 creates variables for all input and output streams of the component specifications on the left side and right side of the specification check. The lower part of the translation rule adds a negation in case the property *negation* of the specification check equals **`true`**. For every component type definition a corresponding predicate is instantiated parametrized with the variables of input and output streams. Please note that in this case the order of the variables is important: for the translation rule we assume that the elements $p \in cmp.CPorts$ and $val \in p.type$ are iterated in the same order as in the translation rules C1 shown in Figure 7.25 and A1 shown in Figure 7.32 for the heads of the predicates of composed and atomic MontiArcAutomaton components. Our implementation based on MontiCore guarantees this property.

Finally we include the definition of the variable `allTime` from Listing 7.20 and references to the files containing the generated Mona predicates for the component type definitions in the specification check. The Mona program is then executed by the Mona tool and we interpret the outcome of the execution to decide the result of the specification check. Please note that the outcome of a Mona program can be of the three kinds **`tautology`**, **`contradiction`**, or **`examples`** (see Section 7.3.1). In case the specification check does not include negation the verification result is positive if the Mona outcome is of the kind **`tautology`**.

The use of negation as shown in the translation rule SC from Figure 7.56 is a bit more involved. All stream variables and the set `allTime` in Mona are implicitly universally quantified. The negation in rule SC is only placed behind the universal quantification while the analysis problem requires it in front of the quantification. The formula checked by Mona for the negated equality or refinement problem Φ is $\forall i \in \vec{I}^*, o \in \vec{O}^* : \neg\Phi$ when the original problem is $\neg(\forall i \in \vec{I}^*, o \in \vec{O}^* : \Phi)$. The latter is however equivalent to $\exists i \in \vec{I}^*, o \in \vec{O}^* : \neg\Phi$. Thus, if Mona determines the kind **`examples`** or **`tautology`** as

Translation Rule

Translation rule:

SC *variables for streams on all input and output ports of left side components*
$\forall p \in \bigcup_{cmp \in LeftSide \cup RightSide} cmp.CPorts$:
var2 $\{\!|p.name\_val|\!\}'_{val \in p.type}$;
streams restricted to all points in time
assert (($\{\!|p.name\_val|\!\}^{union}_{val \in p.type}$) sub allTime) ;
$\forall v_1 \in p.type$: *at most one value at any point in time*
assert ($p.name\_v_1$ inter ($\{\!|p.name\_v_2|\!\}^{union}_{v_1 \neq v_2 \in p.type}$) = empty) ;
if (*negation*) **then** $\sim$
specification conjunction on left side of check
({ (*cmp.cType* ({{$p.name\_val$}$'_{val \in p.type}$}$'_{p \in cmp.CPorts}$) }$^{\&}_{cmp \in LeftSide}$
if (*rel* = refines) **then** => **else** <=>
specification conjunction on right side of check
{ *cmp.cType* ({{$p.name\_val$}$'_{val \in p.type}$}$'_{p \in cmp.CPorts}$) }$^{\&}_{cmp \in RightSide}$) ;

Result of application to check `Spec1bRefinesSpec1a` from Listing 7.54:

Mona

```
1  var2 bump_false;
2  var2 bump_true;
3  assert( (bump_false union bump_true) sub allTime );
4  assert( bump_false inter (bump_true) = empty );
5  assert( bump_true inter (bump_false) = empty );
6  similar for rMot_FORWARD, rMot_BACKWARD, and rMot_STOP
7  similar for lMot_FORWARD, lMot_BACKWARD, and lMot_STOP
8
9  (bumperbot_BumpControlSpec1b(bump_false, bump_true,
10   rMot_FORWARD, rMot_BACKWARD, rMot_STOP, lMot_FORWARD,
11   lMot_BACKWARD, lMot_STOP, allTime) =>
12 bumperbot_BumpControlSpec1a(bump_false, bump_true, rMot_FORWARD,
13   rMot_BACKWARD, rMot_STOP, lMot_FORWARD, lMot_BACKWARD,
14   lMot_STOP, allTime));
```

Figure 7.56.: Translation rule SC for MontiArcAutomaton specification checks from Definition 7.55 into Mona.

outcome for the negated case the original problem is positively verified. In case of the outcome **`contradiction`** the assertion of the check is indeed false.

7.4.3. Witnesses for Non-Satisfaction

In the case that the generated formula is not a tautology Mona computes a shortest counter example. The shortest counter example is an assignment of all free variables such that the formula is not satisfied. In the case of refinement checking this means that the assignment satisfies the predicate of the implementation but not the predicate of the specification.

An excerpt of the Mona output for executing the check `BumpControl` **`refines`** `BumpControlSpec1c` is shown in Listing 7.57.

MonaAnalysisOutput

```
1  A counter-example of least length (3) is:
2
3  allTime = {0,1,2}
4  bump_false = {}
5  bump_true = {0,1}
6  ts_ALERT = {}
7  tc_DELAY = {}
8  tc_DOUBLE_DELAY = {2}
9  tc_ABORT = {}
10 rMot_FORWARD = {1}
11 rMot_BACKWARD = {2}
12 rMot_STOP = {0}
13 lMot_FORWARD = {1}
14 lMot_BACKWARD = {2}
15 lMot_STOP = {0}
```

Listing 7.57: A counter example computed by Mona when checking `BumpControl` **`refines`** `BumpControlSpec1c`.

The set `allTime` is shown in line 3 of Listing 7.57 and contains the natural numbers up to 2. Thus, two transitions have been executed for the inputs at time 0 and time 1 to determine the output up to time 2. The variables `bump_false` and `bump_true` in lines 4-5 describe the prefix of the input stream on the port `bump` of the component `BumpControl`. The variable assignments shown in Listing 7.57 can be translated into a more readable format by constructing the message streams represented by the variables. The witness that the component `BumpControl` does not refine the component `BumpControlSpec1c` is then given as:

$$
\begin{aligned}
bump &= \langle \mathbf{true}, \mathbf{true}, \varepsilon_{\text{,}} \rangle \\
ts &= \langle \varepsilon_{\text{,}}, \varepsilon_{\text{,}}, \varepsilon_{\text{,}} \rangle \\
tc &= \langle \varepsilon_{\text{,}}, \varepsilon_{\text{,}}, \texttt{DOUBLE\_DELAY} \rangle \\
rMot &= \langle \texttt{STOP}, \texttt{FORWARD}, \texttt{BACKWARD} \rangle \\
lMot &= \langle \texttt{STOP}, \texttt{FORWARD}, \texttt{BACKWARD} \rangle
\end{aligned}
$$

This example shows a behavior of the component `BumpControl` that is not allowed by the component `BumpControlSpec1c`: whenever the component `BumpControlSpec1c` receives the message **`true`** on the port `bump` it requires the activation of the bumper bot by sending the message `FORWARD` on both motor ports. In this case the specification is too strict to allow different behavior once the bumper bot has been activated.

7.5. Advanced Analyses Example

We continue the example from Section 7.1 of the engineering team developing the software for a bumper bot.

The engineering team has developed a software for the bumper bot with the emergency stop feature based on the component `BumpControl` from the regular bumper bot. The extended version uses an arbiter component that is connected to the emergency stop button and bypasses the component `BumpControl` with a constant stop command when the emergency switch is toggled. An apprentice has been ordered to do a redesign of the component as a single automaton. The result of the work is component `BumpControlES2` shown in Figure 7.58.

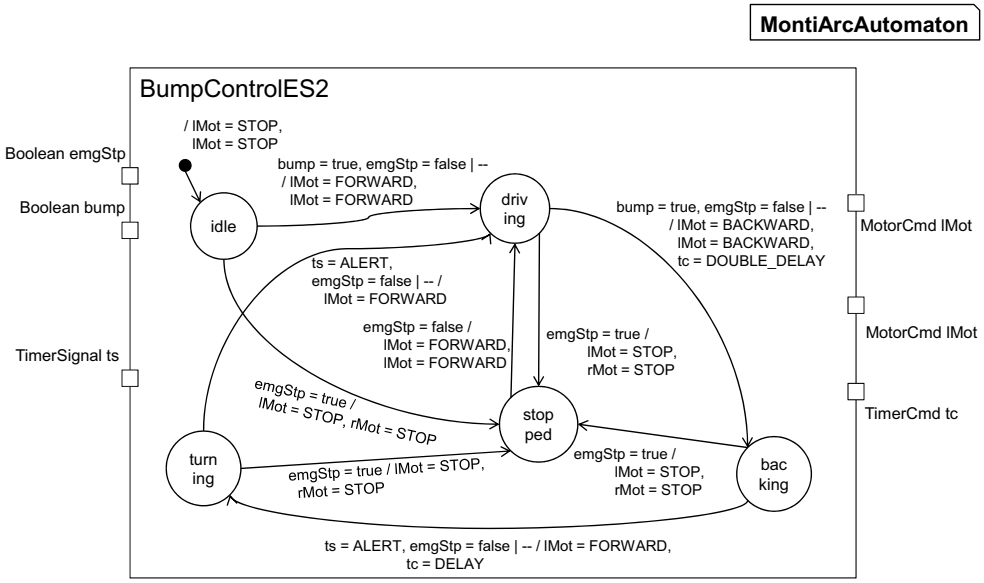


Figure 7.58.: An implementation of the controller for the bumper bot with the emergency stop feature. The automaton is proposed as an alternative to the composed component `BumpControlES` from Figure 7.10.

The automaton of component `BumpControlES2` is based on the automaton of component `BumpControl` from Figure 6.3. The apprentice added the state `stopped` and transitions from every original state of the automaton. The new transitions are enabled when the value **`true`** is received on the input port `emgStp`. In addition, the apprentice updated the other transitions to only being enabled if the port `emgStp` received either the message **`false`** or no message. The alternative is expressed using the symbol `|` and the symbol `--` denoting that no message is received (ε). Finally, the apprentice added a resume functionality: if the automaton is in the state `stopped` and receives the message **`false`** on the port `emgStp`, it changes to state `driving` and sends the `FORWARD` command on both `MotorCmd` output ports.

The quality inspector checks the new design to verify that it complies with the specification for the bumper bot with the emergency stop feature. She executes the MontiArcAutomaton specification suite shown in Listing 7.59 that checks whether the new implementation `BumpControlES2` refines the three specifications presented in Section 7.1.

MAASpecification

```
1  suite EmergencySpecs {
2
3    check BumpControlES2refinesSpec1a:
4      // does not start unless bumper is initially pressed
5      BumpControlES2 refines BumpControlSpec1a;
6
7    check BumpControlES2refinesSpec1b:
8      // FORWARD commands are sent if bumper is initially pressed
9      BumpControlES2 refines BumpControlSpec1b;
10
11   check BumpControlES2refinesSpec2:
12     // STOP commands are sent if emergency button pressed
13     BumpControlES2 refines BumpControlESSpec2;
14 }
```

Listing 7.59: The MontiArcAutomaton specification suite `EmergencySpecs` consisting of three MontiArcAutomaton specification checks.

The quality inspector is pleased to see that at least the emergency off feature is implemented correctly. The last check in Listing 7.59, ll. 11-13 succeeds and thus the implementation always stops the motors if it receives the message **`true`** from the emergency stop button on the port `emgStp`.

The other two specifications are not satisfied. The quality engineer reviews the counter example produced by Mona that proves that the implementation does not refine the specification by component `BumpControlSpec1b` as asserted in Listing 7.59, ll. 7-9. The specification from Figure 7.5 requires the bumper bot to send `FORWARD` commands to the motors if activated by an initial pressing of the bump sensor. The generated counter example consists of the prefixes of the streams on all input ports: $bump = \langle \textbf{true} \rangle$, $emgStp = \langle \textbf{true} \rangle$, and $ts = \langle \varepsilon \rangle$. It reveals that component `BumpControlES2` violated

the specification because the emergency stop was activated. The quality inspector needs to include this case in the specification, which was created before the emergency stop feature was known.

Finally, the quality inspector is surprised that the simple requirement of the bumper bot to not start moving before the bumper is initially pressed is also violated (see Listing 7.59, ll. 3-5). She reviews the model of the implementation provided by the apprentice in Figure 7.58. Still puzzled why the specification is violated the quality inspector reviews the counter example generated by the verification. The counter example is: $bump = \langle \textbf{false}, \textbf{false}, \varepsilon_1 \rangle$, $emgStp = \langle \textbf{true}, \textbf{false}, \varepsilon_1 \rangle$, and $ts = \langle \varepsilon_1, \varepsilon_1, \varepsilon_1 \rangle$. Thus, the verification has exploited the emergency stop feature in combination with the resume feature to activate the bumper bot without the initial pressing of the bump sensor. This problem is serious since it might lead to unwanted activation of the robot. The apprentice needs to fix the implementation.

7.5.1. Replacement of Components

The engineering team decides to replace the software implementation of all bumper bot variants by a single implementation. Thereby, for both variants only one implementation needs to be maintained. The component `BumpControlES2` shown in Figure 7.58 seems to be a good candidate to replace the component `BumpControl` in the bumper bot without the emergency stop feature.

The chief engineer only agrees to the change if the apprentice can show that the replacement ensures equal behavior for the bumper bot without the emergency feature. The apprentice checks the two components `BumpControl` shown in Figure 6.3 and `BumpControlES2` shown in Figure 7.58 for equality as shown in lines 3-4 in Listing 7.60. The verification tool reports different behavior since the automaton of component `BumpControlES2` sends `STOP` messages to the motors if the emergency stop button is pressed while the component `BumpControl` ignores the emergency stop input.

MAASpecification

```
1 suite ReplacementChecks {
2
3   check ComponentEquality:
4     BumpControl equals BumpControlES2;
5
6   check CompositionEquality:
7     BumperBotApp equals BumperBotAppV2;
8 }
```

Listing 7.60: The MontiArcAutomaton specification suite `ReplacementChecks` consisting of two MontiArcAutomaton specification checks.

The apprentice is unhappy with the result and wants to check the behavior of the component `BumpControlES2` in the context of the bumper bot system. She creates a

model that composes all *application* components of the bumper bot, i.e., all its components except for sensors and actuators. The application part of the bumper bot is shown in Figure 7.61 (a). The apprentice also creates a second variant of the application part but with the component `BumpControl` replaced by the component `BumpControlES2` shown in Figure 7.61 (b). In this model the input port `emgStp` of the component `BumpControlES2` is not connected.

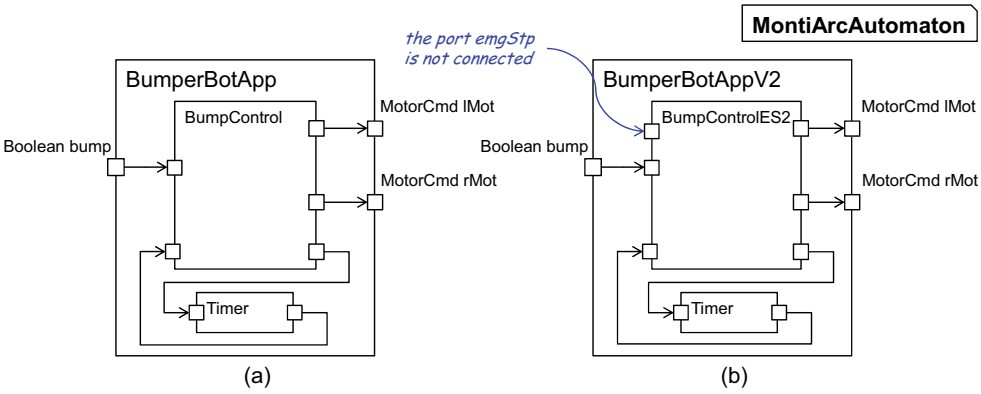


Figure 7.61.: Two versions of the *application* part of the C&C architecture of the bumper bot. The application part consists of all components of the robot except for the sensors and actuators.

Finally, the apprentice verifies that the two compositions have equivalent behavior and that it is safe to replace the component `BumpControl` with the component `BumpControlES2` (Listing 7.60, ll. 6-7).

7.5.2. Specification of the Environment

A new quality inspector has joined the team. She is surprised to see that the most important specification for the bumper bot had not been defined and verified so far:

- [Spec3] The bumper bot drives backwards whenever an obstacle is hit.

The specification is implemented as a MAA_{ts} automaton shown in Figure 7.62. The initial state of the specification is the state `idle` that is left after the initial pressing of the touch sensor. In the state `driving` the component sends `BACKWARD` messages whenever an obstacle is hit.

The quality inspector creates a MontiArcAutomaton specification check to verify that the implementation of the component `BumpControl` satisfies the specification of the component `BumpControlSpec3`. The check is shown in lines 3-4 of Listing 7.63. It is not satisfied and a witness is generated. The quality inspector reviews the witness generated by our tool to understand the cause of the violation. The prefixes are

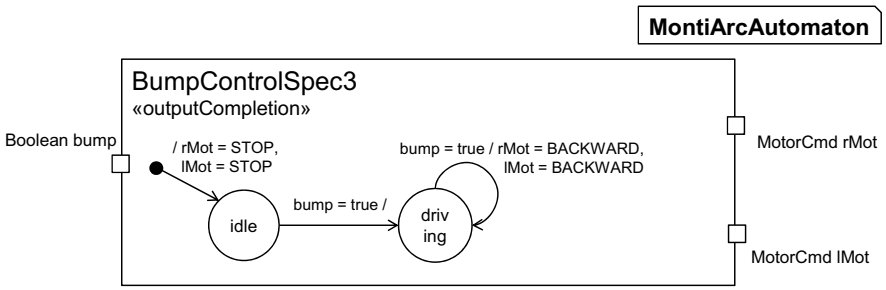


Figure 7.62.: A specification for the bumper bot to drive backwards whenever the bumper is pressed.

$bump = \langle \mathbf{true}, \mathbf{true}, \mathbf{true} \rangle$ and $tc = \langle \varepsilon, \varepsilon, \varepsilon \rangle$. The first message **true** received on the port `bump` activates the bumper bot. The second message **true** starts the backwards driving process. The third message **true** is ignored by the automaton inside the component `BumpControl` shown in Figure 6.3. The engineers did not expect the front bumper to be pressed while the robot is driving backwards. The assumption is reasonable.

MAASpecification

```
1 suite BumperBotBacks {
2
3   check BackingBumperBot:
4     BumpControl refines BumpControlSpec3;
5
6   check BackingWithEnvBumpGuard:
7     (BumpControl and EnvBumpGuard) refines BumpControlSpec3;
8 }
```

Listing 7.63: The MontiArcAutomaton specification suite `BumperBotBacks` consisting of two MontiArcAutomaton specification checks.

To solve this problem the engineering team decides that the specification should include some knowledge about the environment. The current analysis based on Mona evaluates the MontiArcAutomaton specification check for all possible input streams on the ports `bump` and `tc` of the component `BumpControl`. These possible values should be restricted to the ones that can occur in realistic environments of the bumper bot.

The team develops the guard component `EnvBumpGuard` for the environment. This component reads the commands that the component `BumpControl` sends to the motors and evaluates whether the current driving behavior allows bumping into objects. When the robot is not driving forward the input `bump` must be set to **false**. The updated specification check is shown in Listing 7.63, l. 6-7. The implementation `BumpControl` with the environment guard `EnvBumpGuard` refines the specification `BumpControlSpec3`.

The MontiArcAutomaton implementation of the environment guard is shown in Figure 7.64 (a). The composed component `EnvBumpGuard` consists of two instances of the component `MotorObserver`, each connected to one of the `MotorCmd` input ports `lMot` and `rMot`.

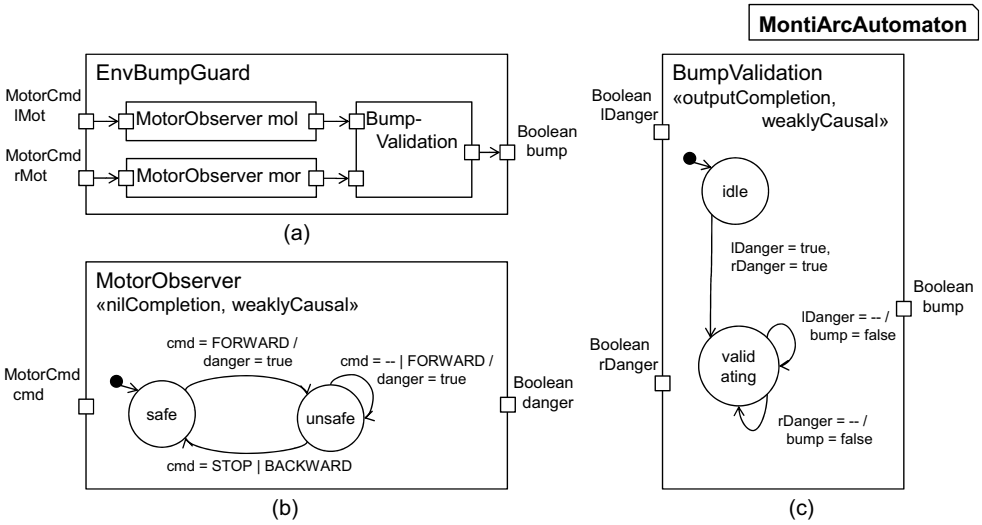


Figure 7.64.: The implementation of the environment guard `EnvBumpGuard` (a). The guard is implemented as a composition of MAA_{ts} automata.

The component `MotorObserver` is shown in Figure 7.64 (b). The component has two states and decides based on input of the type `MotorCmd` whether the motor drives forward or not. If the motor command `FORWARD` is received, the automaton of the component switches to the state `unsafe` and sends the message **`true`** on the outgoing port `danger`. The message **`true`** is repeated on the port `danger` as long as the motor turns forward. If the motor is stopped or turns backward nothing is sent on the port `danger` (see the stereotype «`nilCompletion`»).

The component `BumpValidation` is shown in Figure 7.64 (c). Inside the composition defined by the component `EnvBumpGuard` the instance of the component `BumpValidation` evaluates the messages sent by the `MotorObserver` components of the left and right motor. The automaton inside component `BumpValidation` switches from the state `idle` to the state `validating` if it receives the message **`true`** on both input ports `lDanger` and `rDanger` that indicate that the motors are going forward (initial activation of the bumper bot). In the state `validating` the automaton sends the message **`false`** on the port `bump` whenever one of the ports `lDanger` or `rDanger` does not receive a message. If one of the `MotorObserver` components does not report danger the bumper bot is either stopped, turning, or driving backwards. The automaton

inside the component `BumpValidation` uses output completion. In this case output completion allows arbitrary outputs on the port `bump` if nothing is specified, i.e., the environment is not restricted unless `BumpValidation` requires the message **`false`**.

Both MAA_{ts} automata shown in Figure 7.64 use the stereotype «`weaklyCausal`» which indicates an immediate reaction of the components to their input. The component `EnvBumpGuard` does not contain feedback cycles and thus weak causality allows a valid composition. The stereotype «`weaklyCausal`» is currently not supported by our implementation. The verification result of this example is based on manual modifications of the translation. For a discussion of the advanced feature of weakly causal MAA_{ts} automata translations see Section 7.7.6.

The MontiArcAutomaton specification check that combines the environment guard, the bumper bot implementation, and the specification from Listing 7.63, ll. 6-7 is defined as (`BumpControl` **`and`** `EnvBumpGuard`) **`refines`** `BumpControlSpec3`. Please note that the conjunction of the implementation `BumpControl` and the specification `EnvBumpGuard` is not a composition of the corresponding MontiArcAutomaton components. The conjunction denotes that input and output streams are restricted by both specifications independently.

7.6. Implementation and Evaluation

We have implemented the translation of MontiArcAutomaton models for composed components, atomic components, and for components with manual Mona implementations into Mona. Our implementation uses the MontiCore template-based code generation framework [Sch12]. In addition, we have implemented a MontiArcAutomaton specification suite verification tool based on an extended version of the MontiArcAutomaton specification suite and check language presented in Section 7.4.1. A screen capture of the verification tool is shown in Figure 7.65.

Our implementation consists of a translation of MontiArcAutomaton components into Mona and a translation and verification engine for MontiArcAutomaton specifications both written in Java. The verification engine invokes the Mona executable using a process call. We report the size of the implementation in effective lines of code (ELOC). Lines counted as ELOC contain characters other than white space or comments and are contained in classes of the implementation. Thus the numbers of ELOC do not include unit tests and code for validation. The translation of MontiArcAutomaton components into Mona consists of 20 classes with a total of 1,263 ELOC and 13 Freemarker templates with 333 ELOC. The translation and verification engine for MontiArcAutomaton specifications consists of 28 classes with a total of 2,138 ELOC and 12 Freemarker templates with 226 ELOC.

A user of the verification tool may define MontiArcAutomaton specification suites consisting of MontiArcAutomaton specification checks. Suites are executed using the front-end of the JUnit testing framework [wwwi]. Our implementation uses MontiCore tools to parse the specification suite file and generates Mona code for the specifications. The Mona code for each specification check is automatically executed in Mona. The out-

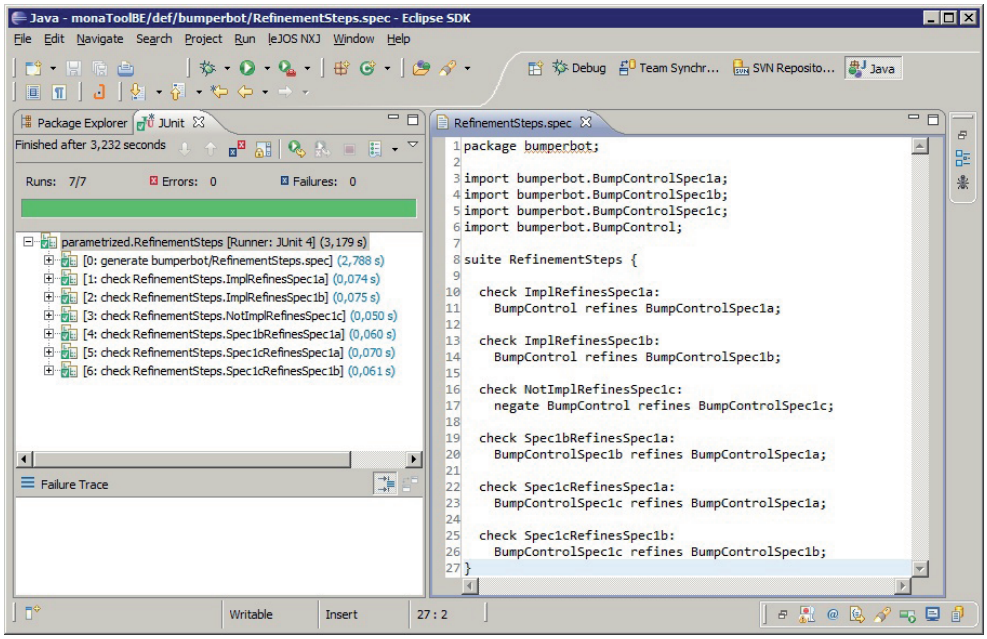


Figure 7.65.: A screen capture of the MontiArcAutomaton specification suite verification tool.

put of the execution is interpreted and the results of passing and failing checks are shown in a JUnit view as demonstrated in the left of Figure 7.65. Counter examples generated by Mona are translated back into prefixes of message streams that show the violation of a specification. For a tutorial on how to create MontiArcAutomaton specification suites and execute the verification using our tools see Appendix E. The implementation with more than 100 component type definitions and 12 MontiArcAutomaton specification suites is available from [wwws].

Based on our implementation of the specification suite verification tool we have created various example systems and specifications. We report on these systems in Section 7.6.1. In Section 7.6.2 we present figures of the execution times of Mona for checking the Mona programs generated by our tool.

7.6.1. Example Systems

We have applied our analysis framework for the verification of MontiArcAutomaton specification suites to three example systems. The evaluation on example systems is of qualitative nature. We wanted to gain experience with creating MAA_{ts} automata for specifying component behavior and analyzing refinement and equality. We report on our

experience in Section 7.7.

The complete set of models described in this section, models from additional experiments, and all MontiArcAutomaton specification suites are available from [wwws].

Bumper bot

We have evaluated our approach on the C&C architecture of a bumper bot. The bumper bot appears as a running example in Chapter 6 and Chapter 7. The bumper bot can power its left and right motors and detect obstacles in front of it. In addition, it is equipped with an emergency stop button. The bumper bot's objective is to avoid obstacles and turn once an obstacle has been hit.

The complete bumper bot example consists of four variants: the basic bumper bot with a component `BumpControl` and two extensions of the robot with an emergency off switch that immediately stops the bumper bot. One of the robots with the emergency stop feature reuses the component `BumpControl` of the original bumper bot. The fourth variant is again a bumper bot without the emergency stop feature that uses the control implementation of a bumper bot that provides the feature.

The complete bumper bot example with all variants consists of 19 component type definitions. Of these components 8 are composed components, 10 are implemented as MAA_{ts} automata, and 1 is implemented manually. The MAA_{ts} automata have at most 3 variables, 3 states, and 10 transitions.

We have defined 4 MontiArcAutomaton specification suites with between 2 and 11 checks. The suites are implementations of the examples presented in Section 7.1 and Section 7.5 and contain additional checks to validate our work. The checks use all of the specification language features presented in Section 7.4.1: negation, refinement, equality, and conjunction of components.

Pump station

We have evaluated the MontiArcAutomaton verification tool on the model of a pump station taken from the AutoFOCUS tool [wwwe, HF07]. This example also appears as a running example in Chapter 3 and Chapter 4. An overview of the components of the pump station is given in Appendix G. The pump station C&C architecture describes the software of a pump station consisting of two water tanks connected by a pipeline system with a valve and a pump. The water level in the first water tank can rise (this is controlled by the environment). When the water level of the first tank rises to a critical level, the water has to be pumped to the second water tank. The second water tank has a drain.

The adapted MontiArcAutomaton model of the pump station consists of 14 component type definitions. Of these components 3 are composed components, 10 are implemented as MAA_{ts} automata, and 1 is implemented manually. The MAA_{ts} automata have at most 3 variables, 3 states, and 26 transitions. We have added 14 component type definitions for refinement and equality checking.

We have defined 3 MontiArcAutomaton specification suites with between 1 and 42 checks. The three suites are mainly created to validate our implementation and to asses the tool's performance on a larger example. The first suite checks all components for equality with themselves. All checks pass. The second suite contains checks for the behavior of some of the pump station components. Finally, the third suite contains checks to assess the performance. For every component of the pump station it checks whether the component refines a specification with an identical interface but only a single state and chaos completion. This suite also contains checks whether the chaos-specification refines the implementation and whether the implementation equals the chaos-specification.

Television set

The television set example is a MontiArcAutomaton model of a television set with a decoder for the commands of a remote control, a volume manager, and a channel manager with two subsystems for checking subscription access rights and for displaying content. The television set was developed as an example in [Kir11].

The complete television set example consists of 12 component type definitions. Of these components 2 are composed components and 10 are implemented as MAA_{ts} automata. The MAA_{ts} automata have at most 5 states and 22 transitions.

We have defined one MontiArcAutomaton specification suite for the television set example with 7 checks. The checks verify specifications for the components of the television set, e.g., the volume manager, the content display and the channel manager. One check verifies a specification of the status output of the complete television set.

Threats to validity

The choice of example systems for our experiments is limited to three systems from different sources. To address this threat to generalizability, we have selected example systems from different domains: robotics, automation, and control.

Limits of the analysis procedure did not allow verifying specification suites for larger examples. This is a threat to the generalizability of our method.

The performance limitations have also constrained us from creating and verifying more complex specifications for the components of the pump station example. For this example system the specification checks are limited to checking equality and refinement with a chaos-specification, which allows arbitrary behavior. More meaningful specifications could eliminate this threat to the qualitative evaluation.

7.6.2. Mona Verification Times

We now address the performance of our prototype implementation for the verification of MontiArcAutomaton specification checks. We are interested in answering the research question whether MontiArcAutomaton specification checking is feasible with our current implementation and whether the verification times are acceptable.

We have measured verification times of Mona on the generated Mona programs for the MontiArcAutomaton specification checks of the bumper bot and the pump station example systems. For each check we report on the number of input and output second order Mona variables of the implementation that is verified against a specification. These variables are universally quantified in the generated Mona programs. The variables encode input and output streams and their number is the sum of the sizes of the port types, e.g., for two output ports of type `MotorCmd` = {`STOP`, `FORWARD`, `BACKWARD`} the number of second order output variables is 3+3 = 6. We have run all checks 12 times and report on the median of the measured running times for each check. All times are given in milliseconds. We performed the experiments on a regular laptop computer, Intel Dual Core CPU, 2.8 GHz, running 64-bit Windows 7 and Java 1.7.0_17. We have used Mona version 1.4-13 compiled for Windows available from [wwwz].

The results of running the bumper bot MontiArcAutomaton specification checks presented in previous examples are shown in Table 7.66. The first column contains the specification name as used in the examples. The second and third columns contain the verified relation and the positive or negative outcome of the check. The following columns give indications of the size of the specification in terms of inputs, outputs, and the number of MontiArcAutomaton component instances including subcomponents.

Each MontiArcAutomaton specification check for the bumper bot is verified in less than 150 ms. The longest verification time is required for the check `BackingWithEnvBumpGuard` which includes a conjunction of the implementation of the component `BumpControl` with the composed component `EnvBumpGuard`.

Check name	relation	result	in #var2	out #var2	#cmps	time (ms)
`Spec1bRefinesSpec1a`	refines	✓	2	6	2	79
`Impl1RefinesSpec1a1b`	refines	✓	3	8	3	125
`NotImplEqualsSpec1c`	equals	✓	3	8	2	103
`BumpControlES2refinesSpec1a`	refines	✗	5	8	2	124
`BumpControlES2refinesSpec1b`	refines	✗	5	8	2	130
`BumpControlES2refinesSpec2`	refines	✓	5	8	2	126
`ComponentEquality`	equals	✗	5	8	2	139
`CompositionEquality`	equals	✓	2	6	6	145
`BackingBumperBot`	refines	✗	3	8	2	118
`BackingWithEnvBumpGuard`	refines	✓	3	8	6	149

Table 7.66.: Running times of the verification of the bumper bot example MontiArcAutomaton specification checks presented in Section 7.1 and Section 7.5.

The results of running the pump station MontiArcAutomaton specification checks are shown in Table 7.67. The first column contains the name of the component type definition. The second column indicates whether the component is implemented as a MAA_{ts} automaton (A), as a composed component type (C), or whether it is manually implemented in Mona (M). The following columns report on the number of inputs, outputs, subcomponents, local variables, states, and transitions where applicable.

The column titled EQ reports on the verification time of a MontiArcAutomaton specification check stating the equality of a component with itself. The verification times range from 59 ms to 211 ms for the verification of the component `PhysicsSimulation` which contains a MAA_{ts} automaton with 26 transitions.

The column titled RC reports on checks whether each component refines a component with the same syntactic interface and an automaton with one state and chaos completion. The verification times are again around 100 ms for many of the small automata. The `ModeArbiter` component type definition requires almost two seconds for the verification and three of the MontiArcAutomaton specification checks do not complete. In the cases for the component type definitions `PhysicsSimulation`, `PumpingSystem`, and `SensorReading` the verification stops after five seconds with a message that Mona ran out of memory. In all of these cases the memory consumption of Mona was close to four gigabytes before Mona aborted the execution.

Finally, the column titled EQC reports on checks whether each of the components equals its single-state-chaos-implementation. This verification takes in total a few milliseconds longer than the check RC. Again, the verification for the component type definitions `PhysicsSimulation`, `PumpingSystem`, and `SensorReading` aborts with an out of memory message from Mona.

Component	kind	in #var2	out #var2	subs	var #var2	#state	#trans	EQ (ms)	RC (ms)	EQC (ms)
`Controller`	C	4	5	3	–	–	–	84	153	168
`EMSOperation`	A	2	5	–	–	1	2	72	78	72
`ModeArbiter`	A	10	5	–	–	1	2	78	1874	2022
`PhysicsSimulation`	A	6	23	–	21	1	26	211	–	–
`PumpActuator`	A	4	2	–	–	1	1	67	130	112
`PumpingSystem`	C	20	6	4	–	–	–	165	–	–
`PumpSensorReader`	A	2	2	–	–	1	1	59	104	87
`SensorReading`	C	20	9	4	–	–	–	139	–	–
`SimulationPanel`	M	23	2	–	–	–	–	119	155	142
`TankSensorReader`	A	5	2	–	–	1	2	75	90	75
`UserButtonReader`	A	2	2	–	2	1	3	65	97	78
`UserOperation`	A	2	5	–	–	1	2	72	82	82
`ValveActuator`	A	6	4	–	–	3	7	74	100	92
`ValveSensorReader`	A	11	3	–	–	1	11	109	124	110

Table 7.67.: Running times of the verification of MontiArcAutomaton specification checks for the component type definitions of the pump station example.

Threats to validity

The choice of example systems is a threat to the generalizability of the findings about feasibility and performance of MontiArcAutomaton specification check verification. To mitigate this threat we have chosen a small example system completely developed in the context of this work and one taken from a public available resource. The systems are different in the number and size of components, which is also reflected by the verification times reported in Table 7.66 and Table 7.67.

The specification checks developed for the pump station example system do not represent realistic examples. For this system only the implementation was available and no specifications expressed as MAA_{ts} automata. We nevertheless performed three basic checks per component for refinement and equality. These checks assess the capabilities of the prototype implementation on specifications with more input variables, more output variables, and more components when compared to the bumper bot example.

7.7. Discussion

In the following, we discuss several aspects and limitations of the current implementation of the analysis framework for MAA_{ts} automata.

7.7.1. Performance

The running times of the specification checks for the example systems presented in Section 7.6.2 show that our current implementation based on Mona handles small examples reasonably fast. For most examples the running times of the verification are between 50 and 150 ms. However, the results of our evaluation indicate that the current solution is not able to handle specification checks where the sum of the values of the input and output ports' types of the implementation is around 30.

One possibility to address the limitation of the input and output sizes is a different encoding of streams in Mona. The encoding introduced in Section 7.3.2 and used in the translation into Mona is rather naive. In the current encoding each value on a stream is represented by one second order variable in Mona. Thus, the Mona encoding of a stream with a type with 16 values results in 16 variables. A more efficient encoding could represent the same stream in a binary encoding using only four variables. We consider optimizations and their evaluation a future work.

7.7.2. Supported Elements of MontiArcAutomaton and $^*MAA_{ts}$ Automata

Our translation of MontiArcAutomaton component type definitions including $^*MAA_{ts}$ automata supports the translation of component type definitions from Definition 6.8 and $^*MAA_{ts}$ automata from Definition 6.20 with the following restrictions. The first restriction is that all types of input ports, local variables, and output ports are required to be finite. To carry out an analysis using Mona the types also may not have many values as demonstrated by the examples in Section 7.6.2. Types are defined as enumerations with enumeration values in UML/P class diagrams.

The translation of $^*MAA_{ts}$ automata supports the elements defined in Definition 6.20 except for guard predicates on transitions. In addition to syntactic underspecification using the symbol $*$ and non-deterministic transitions the translation supports ε, completion, output completion, and chaos completion. Adding support for guard predicates requires the translation of Java or OCL/P guard expressions into Mona.

Additional features of the MontiArcAutomaton modeling language that are not contained in Definition 6.8 of component type definitions are generic component types and

parametrized component types. Our current translation does not support these features. An example of a generic component type is the type `Buffer<T>` from Listing 6.6. A concrete type for the type variable `T` is defined for every instantiation of the component. With our current translation of component type definitions to Mona there is no similar concept to type variables that would allow us to generate a Mona predicate for the component type `Buffer<T>`.

A simple way to support the translation of generic and parametrized component type definitions is the translation of concrete instantiations with concrete parameters unfolded in the definition of the component type. This approach gives up a compositional translation into Mona since generic and parametrized components can only be translated if their type parameters and value parameters are known from subcomponent instantiations.

7.7.3. Language Expressiveness

In our experiments with writing specifications as MontiArcAutomaton components we have found it useful to evaluate refinement with respect to an upward simulation as presented in Section 6.3.4. We consider it important to support the verification of composed components. Implementations created by the composition of components are a crucial part of system development. For specifications the same rationales are be applied. Instead of considering the specification as a single model our approach allows to decompose specifications and reuse existing ones. The syntax and semantics for the composition of specifications are the same as for the composition of components.

An orthogonal concept to the composition of components is the conjunction of specifications and implementations. On the one hand, this allows reuse of specifications and a more succinct definition of MontiArcAutomaton specification checks involving multiple specifications and a single implementation. On the other hand, the conjunction of specifications allows imposing additional restrictions, e.g., on the environment as shown in Section 7.5.2.

During the creation and reuse of specifications for composed component type definitions we have encountered difficulties with handling the accumulated processing delays introduced by component composition (see Section 7.7.5). We have experimented with a modified translation that allows immediate processing. This topic requires further investigation.

7.7.4. Refinement and Equality

Our translation of MontiArcAutomaton models into Mona results in predicates over input and output streams. These predicates express the I/O relation semantics of MAA_{ts} automata defined in Definition 6.33 restricted to all finite prefixes. The Mona encoding of the semantics of MontiArcAutomaton components enables the analysis framework to check component refinement based on input and output streams.

We have defined two notions of refinement for MontiArcAutomaton components. Refinement with respect to upward simulation [Bro93], as defined in Definition 7.12, does not require equality of the component interfaces of the implementation and specification.

The implementation may have additional inputs and outputs. The refinement as defined in Definition 7.13 is a more classical version [LS11]. The definition states that the implementation requires at most all inputs of the specification and provides at least the same outputs. In this case the implementation can syntactically replace the specification.

Similarly, we have defined component equality only on shared ports in Definition 7.14 and for equivalent component interfaces in Definition 7.15.

In our current implementation we check refinement and equality on all shared input and output ports. Combinations of these behavior refinement and equality checks with syntactic checks of the component's interfaces allow to determine the results for the cases discussed above. We consider it a future work to extend the specification suite language introduced in Section 7.4 with more specific refinement statements that subsume the syntactic checks.

7.7.5. Component Processing Delay

We have chosen a strongly causal semantics for ${}^{*}\mathrm{MAA}_{ts}$ automata. Strong causality implies that a timed component may react to an input message only in the next time slice. Strong causality has the benefit that the composition of components is well-defined as discussed in Section 6.3.3. A downside of strong causality is however that the sequential composition of components introduces a delay. This delay may be *hidden* inside composed components.

Consider the composed component type `DifferentDelays` shown in Figure 7.68. The component is composed of subcomponents of the component type `Processor`. There is no processing delay between the ports `i1` and `o1` of the component `DifferentDelays` since a connector immediately forwards the messages. The processing delay between the ports `i2` and `o2` is the processing delay of the component type `Processor`. Finally, the delay between the ports `i3` and `o3` of the component `DifferentDelays` is three times the processing delay of the component type `Processor`. For the implementation of systems and subsystems, components are treated as black boxes and respectively no assumptions on the processing delay can be made.

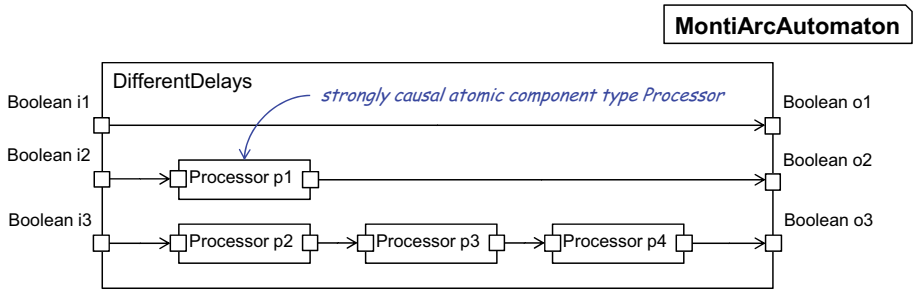


Figure 7.68.: The composed component `DifferentDelays` containing subcomponents of the component type `Processor` with strongly causal behavior.

The accumulated delay introduced by component composition also complicates the specification of component behavior using MAA_{ts} automata. In some cases a specification has to be aware of introduced processing delay. For example the two implementations `BumpControlES` shown in Figure 7.10 and `BumpControlES2` shown in Figure 7.58 differ in the delay of the processing of messages on the port `bump` if the emergency switch is not activated. The composition of the component `BumpControl` with the component `ArbiterMotorCmd` delays the response of the composed component `BumpControlES` by one time slice compared to the response of the component `BumpControlES2`.

7.7.6. Immediate Processing

Components that are only weakly causal instead of strongly causal allow immediate processing of messages. Weak causality is implied by strong causality. For a winning strategy $\tau : \vec{I}^{\infty} \to \vec{O}^{\infty}$ that realizes component behavior weak causality is defined as:

$$\forall \vec{i}, \vec{i}' \in \vec{I}^{\infty} \ \forall n \in \mathbb{N} : \vec{i}|_n = \vec{i}'|_n \Rightarrow \tau(\vec{i})|_n = \tau(\vec{i}')|_n$$

It is possible to adapt the semantics definition of MAA_{ts} automata to allow immediate processing of messages. The adaption of the translation of ${}^{*}\text{MAA}_{ts}$ automata into Mona requires changes in the translation rules of the transition system and the rules for the transition system completions. With optional initial output on ports the transition rules depend on the chosen initial state and its output. In case no initial output is allowed the output of each message on an output port is set to time `t` instead of `t+1` to model immediate processing.

Please recall the example of the component `EnvBumpGuard` that restricts the environment to only allow activation of the bump sensor when the robot moves forward. To carry out the verification, the composed component has to react to input immediately. We have manually changed the time of the outputs on the transitions of the ${}^{*}\text{MAA}_{ts}$ automata inside the components `MotorObserver` and `BumpValidation` shown in Figure 7.64. An excerpt of the code generated for the transitions of the component `BumpValidation` including the changes is shown in Listing 7.69. The difference to the Mona code generated for strongly causal component behavior is in lines 6 and 8. In both cases the term `t in bump_false` defines that the output of the message **`false`** on the port `bump` happens at the same time `t` as the input. In the strongly causal implementation the term was `t+1 in bump_false`.

The generation of Mona code for immediate processing of messages is not fully automated in our current implementation. In addition to the changes in the translation rules, immediate processing also requires additional well-formedness rules for component composition. Every directed communication cycle must contain at least one strongly causal component to allow component composition.

Mona

```
all1 t: t+1 in allTime =>
(
  ( t in idle & t in rDanger_true & t in lDanger_true &
    t+1 in validating ) |
  ( t in validating  & t notin (lDanger_false union lDanger_true) &
    t+1 in validating & t in bump_false ) |
  ( t in validating & t notin (rDanger_false union rDanger_true) &
    t+1 in validating & t in bump_false )
)
```

Listing 7.69: An excerpt of the modified transition system of the component `BumpValidation` presented in Figure 7.64 (c) to support immediate processing of input messages.

7.7.7. Choice of Mona

Mona is an implementation of a decision procedure for the WS1S logic used as a target formalism for our translation of MontiArcAutomaton components. On the one hand we have demonstrated that $^*\text{MAA}_{ts}$ automata can be expressed in WS1S logic and that many analysis problems can be formalized and solved using the Mona implementation. On the other hand the complexity of solving WS1S decision problems is non-elementary [Mey75], i.e., bounded by a stack of exponentials with the height of the size of the formula. The translation of MAA_{ts} automata into WS1S and solving the analysis problems using Mona creates an analysis overhead that needs to be discussed.

For the purpose of this work we have decided to use an implementation based on WS1S and Mona because of the natural formulation of analysis problems that coincides with the formal definitions of the FOCUS framework [BS01]. The decision procedure as well as counter example computation are fully automated. We believe that this trade-off between implementation time and complexity of the automated analysis is legitimate for a research prototype.

Preliminary experiments presented in Section 7.6.2 demonstrate that the current prototype handles many of the examples presented in this thesis in 50-150ms. However, it already fails to produce verification results on other small examples. Please note that these restrictions are due to the complexity of the target logic and not due to the complexity of the analysis problem itself.

Possible future work is the investigation of a translation of $^*\text{MAA}_{ts}$ automata into SMV [BCM+92, wwww] modules and an implementation of the analysis algorithms using BDD-based algorithms for checking trace containment and trace equivalence. Another alternative to evaluate is an encoding in the Promela language of the model checker SPIN [Hol04]. It is important to note that besides a faithful representation of the semantics of MAA_{ts} automata and component compositions an alternative solution also requires the formulation of the analysis problems for MontiArcAutomaton specification checks. The direct formulation of the analysis problems is one of the benefits of Mona.

7.8. Related Work

In this section we discuss three kinds of related works. We briefly review related applications using the verification tool Mona. We then list some approaches that address the analysis of FOCUS specifications by formalization of FOCUS' semantics in various tools. Finally, we discuss related verification approaches and languages.

7.8.1. Mona for Analyses of Focus Specifications and Automata

Our encoding of message streams is inspired by the encoding of message streams by Schätz [Sch09]. Schätz describes the behavior of a software system and subsystem as the composition of modular functions. These functions are formalized as predicates on streams similar to the predicates we generate for components. Schätz also formalizes function composition and refinement in Mona. Modular functions may represent transitions of automata but transition system completions as presented in our work are not discussed. Different from our work the translation of models and specifications into Mona is not automated.

Schätz and Pfaller [SP10b, SP10a] define component behavior using predicates in Mona similar to our formalization. In addition they define a component test case as a sequence of messages received and sent by a component. Schätz and Pfaller use Mona to synthesize the test case for a composed component from a test case for one of its subcomponents. We believe that this approach to test case definition and to test case synthesis can directly be applied to our formalization of MontiArcAutomaton components in Mona.

Hune and Sandholm [HS00] use Mona for the synthesis of controllers for Lego robots. The synthesis approach has as input an implementation of the controller given as an automaton and a specification that restricts the implementation. From these inputs Mona computes a minimal deterministic automaton with behavior that satisfies the specification. The work of Hune and Sandholm [HS00] focuses on synthesis while we focus on verification. Their specifications are logic-based and not modeled as automata. In contrast to our work the approach does not handle composed components but only single automata. It would be interesting to combine this synthesis approach with our translation of automata to logic formulas as input for the synthesis.

Klarlund et al. [KNS96a, KNS96b] have developed the programming language FIDO with a translation of FIDO programs into Mona programs. The language is developed for the specification of properties on recursive data structures, e.g., message streams. In [KNS96b] the authors present a case study of the RPC-memory specification problem proposed by Broy and Lamport [BL94].

7.8.2. Tool Support for the Verification of Focus Specifications

The AutoFOCUS tool [HSSS96, HS97, BHS99, HF07] for the specification and prototyping of distributed systems is based on the FOCUS method and offers graphical representations of modeling and specification artifacts. The behavior of components is modeled

using state transition diagrams similar to MontiArcAutomaton models. To the best of our knowledge none of the verification approaches developed for AutoFOCUS covers the completions defined in [Rum96] and Section 6.4.

Huber et al. [HSE97] report on automated verification of the refinement of AutoFOCUS components described by state transition diagrams and a sequence diagram notation using the model checkers SMV [BCM+92] and μ-cke [Bie97]. Similar to our approach refinement is based on trace inclusion.

Recent works have formalized parts of FOCUS in the interactive theorem prover Isabelle [NPW02]. All of these works formalize notions of streams, components, component composition, and refinement.

Spichkova [Spi07] formalized parts of FOCUS in Isabelle/HOL. System specifications can either be translated manually or developed directly in Isabelle/HOL. The implementation covers timed streams and proposes also ways to handle time-synchronous streams. Based on some of the results Trachtenhertz [Tra09] has formalized semantics for the description techniques of AutoFOCUS in Isabelle/HOL. The framework focuses on temporal specifications of functional properties. The work aims at supporting the development process from design phase to an executable specification. As an industrial case study an adaptive cruise control system is formalized.

Gajanovic and Rumpe [GR06, GR07] have presented an implementation of FOCUS streams based on the logic Isabelle/HOLCF [Slo97, Huf12]. The logic Isabelle/ALICE [GR07] formalizes discrete finite and infinite streams as well as infinite timed (event) streams. The work has been extended by Krüger [Krü11] and Raco [Rac13] with case studies and an implementation of stream bundles for the definition of stream processing functions [RR11].

An interesting future work is the combination of partially automated proofs in the interactive theorem prover Isabelle and fully automated methods such as Mona.

7.8.3. Related Verification Tools

The verification problems we solve with the techniques and implementation presented in this chapter have many possible formalizations in related formalisms used in the area of model checking [BK08]. Related formalisms are, e.g., the SMV specification language [McM99] that allows the definition of modules for the SMV model checker [wwww]. SMV supports the definition of state machines, modules, and module compositions. The composition of modules is similar to the composition of predicates in Mona used in our formalization of component composition. Another language that allows the definition of distributes processes similar to components in FOCUS is the Promela language of the model checker SPIN [Hol04]. We believe that these are two candidate languages for alternative formalizations of MontiArcAutomaton models.

In the context of classical model checking of labeled transition systems (LTS) our notion of refinement corresponds to trace inclusion. A stronger notion of refinement is, e.g., simulation [LS11, BK08]. A state s_1 of an LTS simulates a state s_2 of another LTS if for every successor s_1' of s_1 there is a successor of s_2 that has the same label as

s_1' and simulates s_1'. As an extension of the presented analysis framework we consider supporting also refinement checks based on simulation.

Chapter 8.

MontiArcAutomaton Code Generation

The modeling language MontiArcAutomaton allows the modeling of control components and the composition of their structure and behavior to complex systems. In model-based engineering, code generators turn models into executable code of a programming language. To enable the development of interactive C&C systems and their execution, we have developed a code generation framework for MontiArcAutomaton models. The framework generates code for the deployment to robotic platforms.

The code generator takes as input MontiArcAutomaton models with automata that conform to the MAA_{ts} language profile. It generates Java code that can be directly deployed to a Lego robot. Specifically, the target platform used for demonstration and evaluation purposes is the Lego Mindstorms NXT robotics framework [wwwm] with the Java leJOS firmware [wwwh]. An important feature of the code generation for this target is that the execution of the generated code is designed to be faithful to the semantics introduced in Chapter 6.

The code generator is implemented using the MontiCore template-based code generation framework introduced by Schindler [Sch12]. We have reported on the code generation from MontiArcAutomaton models in [RRW13b]. The case study presented in Section 8.4 has been published in [RRW13a].

Chapter outline and contributions

We start with an example for code generation in Section 8.1. Our main contribution presented in this chapter consists of the development of a target platform and a code generator for MontiArcAutomaton models explained in Section 8.2. Section 8.2.1 and Section 8.2.2 contain preliminary information on the leJOS platform and the MontiCore code generator framework from [Sch12]. We present advanced features of our code generation approach in Section 8.3 including the integration of manual component implementations and a simulation of the generated code.

Section 8.4 reports on a case study conducted as a one-semester course for graduate students using the MontiArcAutomaton code generator to develop a robotic coffee service. We discuss the code generator and case study in Section 8.5 and point out related work in Section 8.6.

8.1. Code Generation Example

Consider a scenario where a team of engineers has developed a complete C&C architecture of the control unit of a bumper bot with an emergency stop button as shown in Figure 8.1. Figure 8.2 shows the component type definition `BumperBotEmergencyStop` for the control unit software.

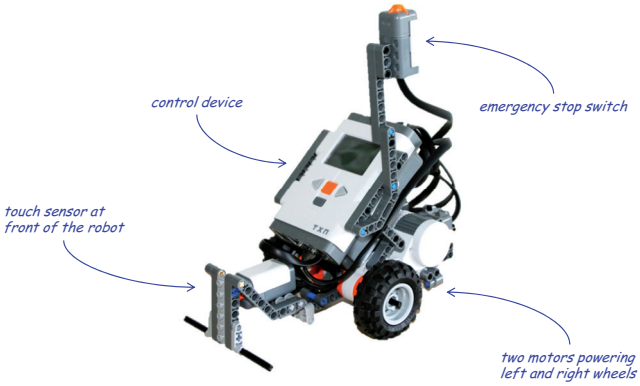


Figure 8.1.: A depiction of the hardware of the bumper bot with an emergency stop switch. The software C&C architecture for the device is shown in Figure 8.2.

The team has created automata implementations for the components `BumpControl`, `MotorStopper`, and `Arbiter<MotorCmd>`. The components for reading sensor data and managing actuators are available with native implementations for the robotics platform the team is using. The component `Timer` is not available yet but one of the engineers will create a manual implementation later. The timer measures system time in milliseconds and can not be implemented as an automaton since it need access to native APIs of system clock on the robotics platform.

An engineer configures the code generator project to use the Lego NXT leJOS library with MontiArcAutomaton models for the components `ToggleSensor`, `TouchSensor` and `Motor`. She then executes the code generator on the MontiArcAutomaton component type definitions of the components shown in Figure 8.2. The code generator generates Java classes for all components and marks the implementation of the component `Timer` with the Java keyword `abstract`.

The leJOS API expert of the engineering team creates the required implementation of the abstract class for the component `Timer`. The generated code of the abstract class `Timer` already provides APIs for accessing the data on input ports and sending the messages via output ports. The engineer simply implements a `compute()` method, which is executed every time cycle.

The software implementation of the bumper bot is ready for deployment on the NXT brick. In addition to the Java classes for the MontiArcAutomaton components the code generator generates a main class that can be directly executed on the Lego NXT robot.

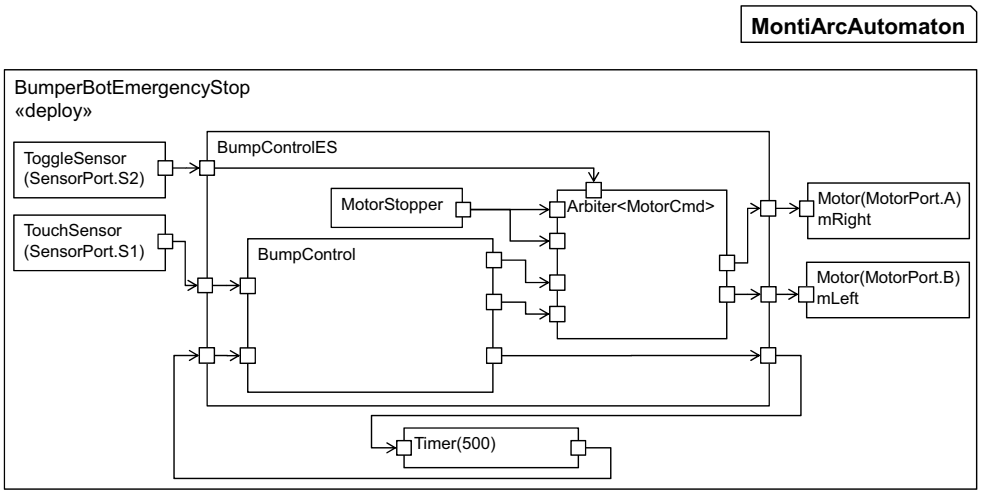


Figure 8.2.: The component type definition `BumperBotEmergencyStop` and its subcomponent type definitions of the bumper bot robot. This figure integrates Figure 7.8 and Figure 7.10.

8.2. MontiArcAutomaton Java Code Generator

This section gives an overview of the code generated by the MontiArcAutomaton Java code generator. We first introduce the Lego NXT target platform that we generate code for and give an overview of template-based code generation for MontiCore languages as introduced in [Sch12]. Section 8.2.3 explains the general structures for components, ports, and variables in the generated code. The code generated for composed components and ${}^{*}\mathrm{MAA}_{ts}$ automata is described in Section 8.2.4 and Section 8.2.5.

8.2.1. The Lego Mindstorms NXT Platform and leJOS

The Lego Mindstorms NXT platform is a robotics platform introduced by Lego in 2006 replacing earlier Lego robotics platforms. Besides being a toy, the NXT platform is also used in education and research projects [IKL+00, HS00, KA03, NBD09, KJ09].

The central processing unit of NXT robots is the Lego NXT brick shown in Figure 8.3. The NXT brick has limited computing resources with an 32 Bit 48 MHz ARM processor and 64 KB RAM [wwwj]. Each brick has three sockets to connect actuators and four sockets to connect sensors. The sensors available with the standard education kit are touch sensors, ultrasonic distance sensors, color sensors, and sound sensors. The available actuators are servo motors and lamps [wwwj].

The leJOS project [wwwh] provides a firmware with a Java virtual machine for the Lego NXT brick. Most of the `java.lang` and the `java.system` libraries are im-

Figure 8.3.: The Lego NXT brick with a 48 MHz processor and 64 KB RAM.

plemented for the NXT. The firmware and tools allow the programming of robotics applications in the programming language Java and provide an application programming interface (API) for the sensors and actuators available with the Lego NXT education package [wwwh]. One limitation of the leJOS Java implementation is the lack of support for reflection [wwwh]. Due to this limitation, e.g., the Java implementation of the observer pattern [GHJV95] (classes `java.util.Observable` and `java.util.Observer`) and support for annotations as, e.g., used by JUnit [wwwi] are missing. Any implementation that uses reflection mechanisms cannot be linked for the Java virtual machine of the NXT brick.

The MontiArcAutomaton Java code generator we have developed is based on the leJOS project version 0.9.1-beta.

8.2.2. MontiCore Code Generation

The modeling language MontiArcAutomaton is a MontiCore [KRV10] language. MontiCore facilitates the development of domain-specific modeling languages by providing a grammar for language definition and tools for parser generation and symbol table management. MontiCore also provides a context conditions and code generation framework.

MontiCore languages, such as MontiArcAutomaton, are defined by context-free grammars. To specify and check well-formedness rules not expressible in context-free grammars, e.g., whether a variable is defined twice, MontiCore provides a compositional Java-based context condition framework [Vö11]. As these context conditions often require information from other models (e.g., to determine whether an assignment violates a type constraint), MontiCore also contains a compositional symbol table framework [Vö11], to facilitate development of complex context conditions.

The component diagram in Figure 8.4 illustrates the components of the MontiCore DSL framework: based on the MontiArcAutomaton grammar MontiCore generates a

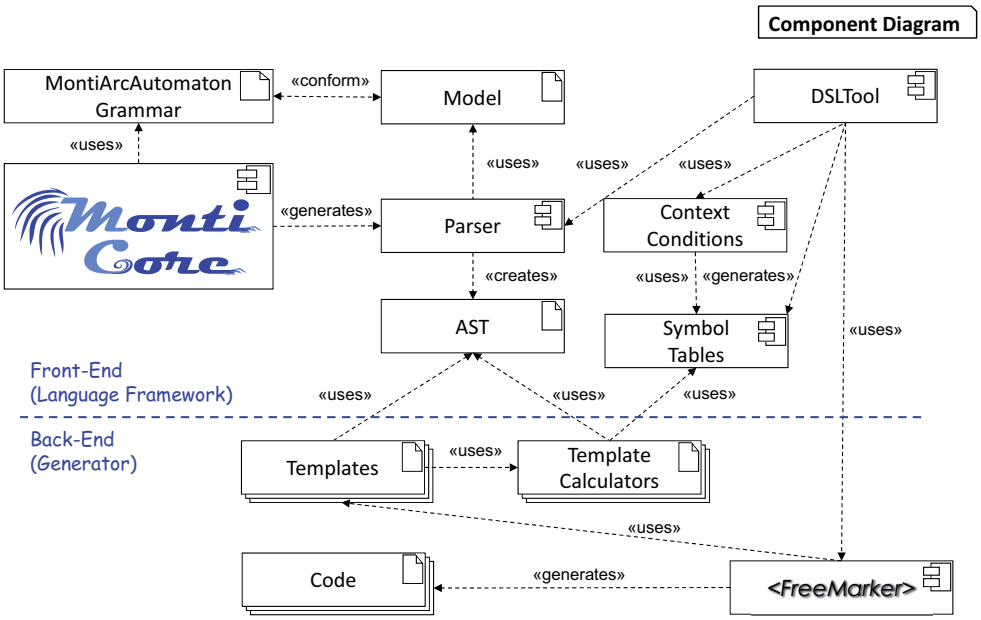


Figure 8.4.: MontiCore uses the grammar to generate a parser for MontiArcAutomaton models which creates the AST (see [Sch12]). The DSLTool uses the parser to read models, which are validated by the context condition framework using the symbol tables provided by the DSLTool. The DSLTool further may use FreeMarker templates and template calculators to generate code from the models based on the AST.

parser for MontiArcAutomaton models. The MontiCore DSLTool executes the parser on a model and orchestrates the generation of symbol tables and the checking of context conditions on the abstract syntax tree (AST) generated by the parser. All MontiArcAutomaton context conditions are documented in [RRW14]. After the validation of the AST the DSLTool invokes the FreeMarker [wwwf] template engine for code generation.

The code generation framework of MontiCore extends the Java-based template engine FreeMarker with calculators and MontiCore specific APIs [Sch12]. The templates of the MontiArcAutomaton Java code generator consist of Java code and FreeMarker control directives. The templates have direct access to the contents of the MontiArcAutomaton AST and are typically organized along the AST structure. For complex computations and operations that require information from other models, the templates invoke template calculators. Template calculators are written in Java and may resolve information about related models via the MontiCore symbol table framework. For example, to generate code for *MAA$_{ts}$ automata transitions, a calculator uses the symbol table to look up port types defined in UML/P class diagrams.

8.2.3. MontiArcAutomaton Components, Ports, and Variables in Java

The MontiArcAutomaton Java code generator takes as input a set of MontiArcAutomaton component type definitions and generates a set of Java files. The code generator is specific to the MAA_{ts} profile for time-synchronous communication and the target language Java. One requirement for the development of the code generator was preserving the composition mechanisms of the modeling language MontiArcAutomaton. The code generated for a component should be independent of the use of the component. This enables incremental code generation and the packaging of generated component implementations in component libraries.

For every component type definition the code generator generates a Java class that represents the component. Figure 8.5 shows the interface `Component` that the class generated for a component type definition implements. The interface allows a uniform handling of components and contains four methods that every component is required to implement. The methods `setUp()` and `init()` are called only once for the instantiation of the component. The methods `compute()` and `update()` are called in compute and update cycles during the execution of the modeled system.

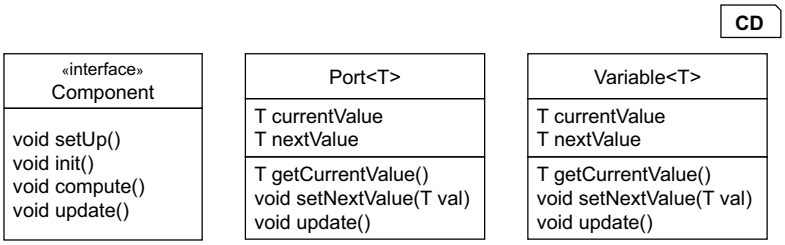


Figure 8.5.: The interface `Component` that allows uniform handling of components and the generic classes `Port<T>` for component ports and `Variable<T>` for local variables of components.

The class `Port<T>` shown in Figure 8.5 represents MontiArcAutomaton ports in the generated Java code. The generic class is parametrized with the Java or UML/P [Rum11, Sch12] type of a MontiArcAutomaton port. The Java port implementation is specific to the time-synchronous semantics of the MAA_{ts} language profile. Because of strongly causal time-synchronous communication, messages are only required to be buffered for one time slice. The class `Variable<T>` shown in Figure 8.5 represents MontiArcAutomaton variables in the generated Java code. Similar to ports the values of variables depend on the current time slice and are thus buffered.

The types of the ports and variables in MontiArcAutomaton models may be native Java types or types defined in UML/P class diagrams. As part of the code generation process the MontiArcAutomaton code generator uses the UML/P class diagram code generator presented in [Sch12] to generate Java implementations of UML/P types.

The execution of the generated Java implementations of MontiArcAutomaton compo-

nents has two phases that are continuously executed. First, the `compute()` methods of all components are executed. The components read their input and compute their output for the next time cycle. Second, the `update()` methods of all component are executed and the values on all ports and of all variables are updated to the values for the next time cycle. This division of the execution cycle into a compute and an update phase has the benefit that the order of the execution of the components does not matter for the data available on output ports and local variables. During each computation cycle all values on output ports are fixed until they are replaced in the update phase. The sending and buffering of copies of messages transmitted between the components in a single Java virtual machine is thus not required. This allows us to save valuable resources on the Java virtual machine of the NXT brick.

It is enough if the source port of a connector provides the message for all receiving ports. Figure 8.6 illustrates all instances of ports necessary for the communication of the components of the bumper bot when the component `BumperBotEmergencyStop` is deployed on the NXT brick. Only nine of the total 25 ports are instantiated at runtime. The connected ports only have references to the port instances.

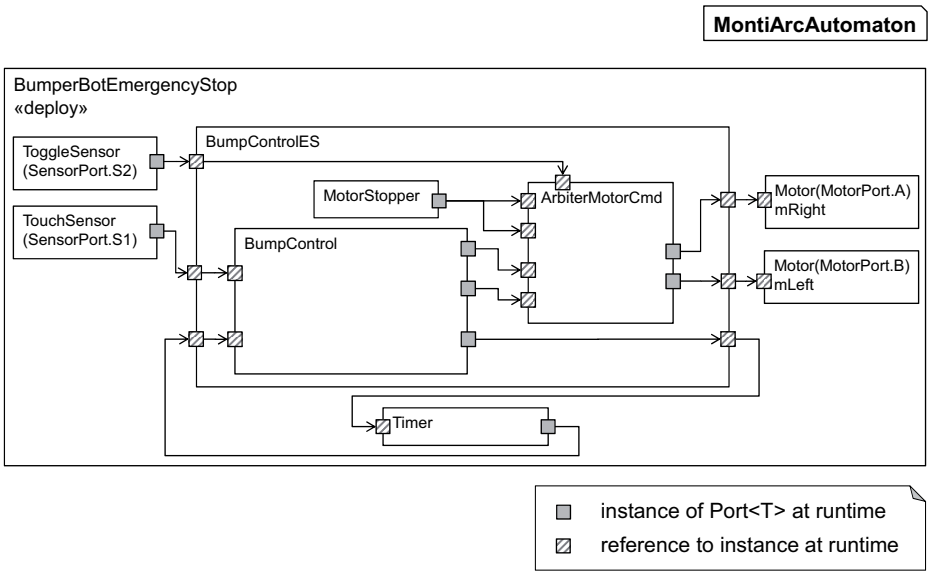


Figure 8.6.: Illustration of the necessary instances of ports (9 of 25 ports) at the runtime of the generated Java code. The dashed ports are only references to port instances.

The MontiArcAutomaton Java code generator distinguishes between the code generated for composed component types (see Section 8.2.4), the code generated for atomic components with automata implementations (see Section 8.2.5), and the code generated for atomic components that have manual implementations (see Section 8.3.1).

8.2.4. Code Generation for Composed Component Types

For every composed component type the Java code generator produces a class with the name of the component, e.g., `BumpControlES`, that implements the interface `Component`. The class contains private fields for all input and output ports of the component. For each input port the code generator generates a public mutator method, e.g., `setPort_bump(Port<Boolean> p)`, to set the instance of the port for the component. For each output port the code generator creates an accessor method, e.g., `getPort_lMot()`, to retrieve the instance of the port. In addition, the code generator creates a private field in the class for every subcomponent. An example of the generated members of the class `BumpControlES` for the MontiArcAutomaton component `BumpControlES` is shown in Figure 8.7.

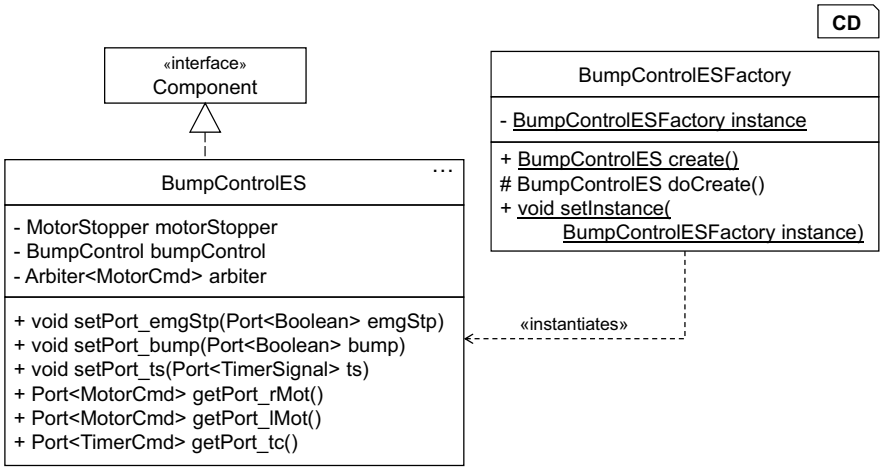


Figure 8.7.: The Java classes `BumpControlES` and `BumpControlESFactory` generated for the MontiArcAutomaton composed component type definition `BumpControlES` shown in Figure 8.2.

The code generated for composed components is complete. The component can be used as is without further requirements for manual implementation. The methods `setUp()`, `init()`, `compute()`, and `update()` are all implemented.

For every component the code generator creates a factory that implements a factory pattern [GHJV95]. The factory class generated for the component `BumpControlES` is shown in Figure 8.7. The creation of objects of the type `BumpControlES` is handled by the static method `create()` of the factory. The generated factory provides a default creation of `BumpControlES` instances via its method `doCreate()`. To allow the replacement of the factory with another instance, the factory provides a static method `setInstance` with a parameter for the instance of the alternative factory implementation. The use of factories for all component instances allows the easy replacement of

component implementations, e.g., for testing and simulation as described in Section 8.3.3.

Execution phases of composed components

The execution of composed components starts with a setup and initialization of the structure of the component using the methods `setUp()` and `init()`. The system then repeatedly executes the `compute()` and `update()` methods.

setUp() The composed component instantiates all of its subcomponents with instances provided from the subcomponent's factories. It then calls the `setUp()` methods of its subcomponents. Finally, the composed component establishes all child-to-parent connectors by creating a reference of its output ports to the output ports of the subcomponent instances.

After the execution of the `setUp()` method all output port fields of the composed component hold references to port instances.

init() The composed component establishes all parent-to-child and all child-to-child connectors between its subcomponents by setting a reference to the source port from all target ports. The composed component finally executes the `init()` methods of all of its subcomponents.

After the execution of the `init()` method, all connectors of the composed component and its subcomponents are established.

compute() The composed component calls the `compute()` methods of each of its subcomponents.

update() The composed component calls the `update()` methods of each of its subcomponents.

Setting up the connectors of composed components is separated. After the execution of the method `setUp()` output ports of the component and all of its subcomponents have references to instances of ports, while the input ports of composed components are still `null` references. The execution of the `init()` method first establishes the remaining connectors on the level of the composed component before initializing the subcomponents. Thus, when the `init()` method of a subcomponent is executed all input ports of the component have references to port instances.

8.2.5. Code Generation for $^*\text{MAA}_{ts}$ Automata

For every atomic component with a $^*\text{MAA}_{ts}$ automaton implementation the Java code generator produces one Java class with the name of the component implementing the interface `Component` shown in Figure 8.5 and a factory for the creation of instance of the class.

The generated class contains private fields for all input and output ports of the component and mutator and accessor methods as in the case of composed components. Thus, the composition of components is the same for composed and atomic components.

For each local variable of the component the code generator generates an attribute of the type `Variable<T>` where `T` is the type of the variable. An enumeration `State` of the states of the automaton is generated inside the class that represents the component. The current state of the automaton is stored in an attribute `state` of the class.

The code generator implements all methods required by the interface `Component`. We describe the purpose of the generated methods in the robots execution phases below before we focus on the generation of the `compute()` method.

Execution phases of atomic components

The execution of atomic components starts with setup and initialization of the component. The system then repeatedly executes the `compute()` and `update()` methods.

setUp() The atomic component instantiates all of its output ports.

init() The component sets the initial state, the initial variable assignment for all variables, and the initial output values on all output ports.

The generated `init()` method sets the field `currentValue` of the variables and ports to make the values immediately available.

compute() Depending on the current value of the variable `state`, the component determines the first enabled transition that matches the values of the local variables and the inputs. The component executes the transition by setting the value of the attribute `state`, the `nextValue` attributes of the output ports, and the `nextValue` attributes of the variables.

update() The component calls the `update()` methods of each of its variables and ports.

Of these methods we focus on the generation of the Java code for the `compute()` method. It is the most important method to implement the strongly causal time-synchronous semantics of $^*\text{MAA}_{ts}$ automata. Consider the example of the MontiArc-Automaton component `BumpControl` shown in Figure 6.3. From the $^*\text{MAA}_{ts}$ automaton inside the component the code generator generates the Java `compute()` method shown in Listing 8.8. The code in Listing 8.8 has been slightly modified to improve presentation, e.g., by removing checks for `null` values from the conditions of the `if` statements.

For every transition $(s_{src}, \phi, \vec{i}, \vec{v}, s_{tgt}, \vec{o}, \vec{a}) \in \delta$ of the $^*\text{MAA}_{ts}$ automaton of the component `BumpControl` one `if` statement is generated in the `compute()` method shown in Listing 8.8. The condition of the `if` statement is a conjunction that checks the satisfaction of the elements s_{src}, ϕ, $\vec{i}$, and $\vec{v}$.

Java

```
public void compute() {
  if (state == State.idle && bump.currentValue == true) {
    rMot.nextValue = MotorCmd.FORWARD;
    lMot.nextValue = MotorCmd.FORWARD;
    state = State.driving;
  }
  else if (state == State.driving && bump.currentValue == true) {
    rMot.nextValue = MotorCmd.BACKWARD;
    lMot.nextValue = MotorCmd.BACKWARD;
    tc.nextValue = TimerCmd.DOUBLE_DELAY;
    state = State.backing;
  }
  else if (state == State.backing &&
                          ts.currentValue == TimerSignal.ALERT) {
    rMot.nextValue = MotorCmd.FORWARD;
    tc.nextValue = TimerCmd.DELAY;
    state = State.turning;
  }
  else if (state == State.turning &&
                          ts.currentValue == TimerSignal.ALERT) {
    lMot.nextValue = MotorCmd.FORWARD;
    state = State.driving;
  }
}
```

Listing 8.8: The generated `compute()` method of the component `BumpControl` shown in Figure 6.3.

For each transition, the first conjunct in the `if` statement checks that the current state `state` of the automaton corresponds to the source state s_{src} of the transition. Next, a translation of the guard ϕ into a Java expression is evaluated. The following conjuncts check the values on all input ports $p \in cmp.CPorts_{IN}$, if an expected input $\vec{i}_p$ is defined ($\vec{i}_p \neq *$). Finally, conjuncts are added for all variables $var \in cmp.CVars$, if an expected variable value $\vec{v}_{var}$ is defined ($\vec{v}_{var} \neq *$).

The generation of the `if` statement implements the reference removal from Definition 6.22 and the enabledness expansion from Definition 6.24. Reference removal is implemented by directly referencing the `currentValue` Java attribute of the element referenced in the $^*\text{MAA}_{ts}$ automaton. Enabledness expansion is implemented by admitting all possible values for inputs and variable values set to $*$.

If a transition is enabled the generated condition of the `if` statement is satisfied and the `compute()` methods executes the body of the statement. The body of the statement is generated from the elements $\vec{o}$, $\vec{a}$, and s_{tgt} of the transition. For every output port $p \in cmp.CPorts_{OUT}$ with $\vec{o}_p \neq *$ the value or referenced value $\vec{o}_p$ is assigned to the attribute `nextValue` of the Java representation of the output port p. Similarly,

for every variable $var \in cmp.CVars$ with an assignment $\vec{a}_{var} \neq *$ the value or referenced value is assigned to the attribute `nextValue` of the Java representation of the variable var. Finally, the target state is set by an assignment of the value s_{tgt} to the attribute `state` of the generated class.

The assignment of the values $\vec{o}_p$ from the example of the component `BumpControl` is shown in the generated Java code in Listing 8.8. Please note that in lines 8-10 values for all three output ports of the component are assigned. In the body of the `if` statement in lines 19-23, only the output on the port `lMot` is set since no other values are specified on the corresponding transition as shown in Figure 6.3.

An example for the result of the implementation of reference removal as defined in Definition 6.22 is shown in Listing 8.9. The listing shows the `compute()` method generated for the component `Arbiter<T>` which is similar to the component `ArbiterMotorCmd` shown in Figure 7.11. The outputs on the ports `outLeft` and `outRight` reference the input port pairs `in1Left`, `in1Right` and `in2Left`, `in2Right`. The translation of the references is shown in Listing 8.9 lines 3-4 and lines 8-9.

Java

```
1  public void compute() {
2    if (state == State.arbiting && mode.currentValue == true) {
3      outLeft.nextValue = in1Left.currentValue;
4      outRight.nextValue = in1Right.currentValue;
5      state = State.arbiting;
6    }
7    else if (state == State.arbiting && mode.currentValue == false) {
8      outLeft.nextValue = in2Left.currentValue;
9      outRight.nextValue = in2Right.currentValue;
10     state = State.arbiting;
11   }
12 }
```

Listing 8.9: The generated `compute()` method of the component `Arbiter<T>` which is similar to `ArbiterMotorCmd` from Figure 7.11 but parametrized with the type `T`.

Together with the `update()` methods of variables and ports this strategy implements Definition 6.25, Item 2 of $\varepsilon_\circlearrowleft$ completion. The `update()` method of the class `Port<T>` replaces the value `currentValue` with the value `nextValue` and sets `nextValue` to `null` which represents $\varepsilon_\circlearrowleft$ in the Java implementation. If the value of an entity is not set in the body of the generated `if` statement its `nextValue` attribute remains `null`. An execution of the `update()` method of the class `Port<T>` propagates the `null` reference to the attribute `currentValue`. The `update()` method of the class `Variable<T>` replaces the value `currentValue` with the value `nextValue` unless `nextValue` is `null`. In all cases `nextValue` is set to `null`. If a next value is not set, the value of the variable is preserved.

The generated code also implements Definition 6.25, Item 3 of $\varepsilon_{\circlearrowleft}$ completion. In case no transition is enabled, none of the bodies of the `if` statements are executed. The component thus remains in its current state. The execution of the `update()` method executes the `update()` methods of the ports and variables and thus sets all outputs to `null` and preserves variable values.

Every execution of the `compute()` method executes at most one transition of the automaton. This is ensured by connecting the `if` statements with `else` in the generated code, e.g., in Listing 8.8, ll. 7, 13, 19. This implementation resolves nondeterminism by executing the first enabled transition.

8.3. Advanced Code Generator Features

This section explains some of the advanced features supported by the MontiArcAutomaton Java code generator. We explain how to use MontiArcAutomaton components with manual implementations and the code that is generated for deployment of a component on the NXT brick. We also report on a preliminary experiment with a Java based simulator for executing the generated code without a Lego NXT robot.

8.3.1. Code Generation for Manually Implemented Components

Some MontiArcAutomaton components of the C&C architecture of robotic systems require access to low level platform APIs not available in the modeling language MontiArcAutomaton. Examples for these components are wrappers for sensors and actuators as, e.g., the components `TouchSensor` or `Motor` of the example robot shown in Figure 8.2. Another example is the parametrized component `Timer[long delay]` that accesses the system clock to measure time. The code generator can not generate a complete implementation for these components. It thus provides a mechanism to add manual implementations in the target language Java.

The parametrized component `Timer[long delay]` can be instantiated with a parameter `delay` of the type `long`. In the example of the bumper bot shown in Figure 8.2 it is instantiated as `Timer(500)` to alert after delays of 500 milliseconds. The MontiArcAutomaton model of the component type definition `Timer[long delay]` is shown in Listing 8.10.

For every atomic MontiArcAutomaton component without a ${}^*\text{MAA}_{ts}$ automaton implementation, the Java code generator generates an abstract implementation with all elements generated as described in Section 8.2.5 for ${}^*\text{MAA}_{ts}$ automata except for the `compute()` method. These methods are left for implementation by the user. As for all classes generated to represent MontiArcAutomaton components, again a factory is generated. For components that require manual implementation, the method `doCreate()` of the generated factory creates instances of a user provided class as shown in Figure 8.11 for the example of the component `Timer[long delay]`.

The manual implementation supplied by the user is expected to be in a Java package that corresponds to the package of the MontiArcAutomaton model. The name of

MontiArcAutomaton

```
component Timer[long delay] {
  port
    in TimerCmd timerCmd,
    out TimerSignal timerSignal;
}
```

Listing 8.10: The MontiArcAutomaton component type definition of the parametrized component `Timer[long delay]` that requires a manual implementation of the component behavior.

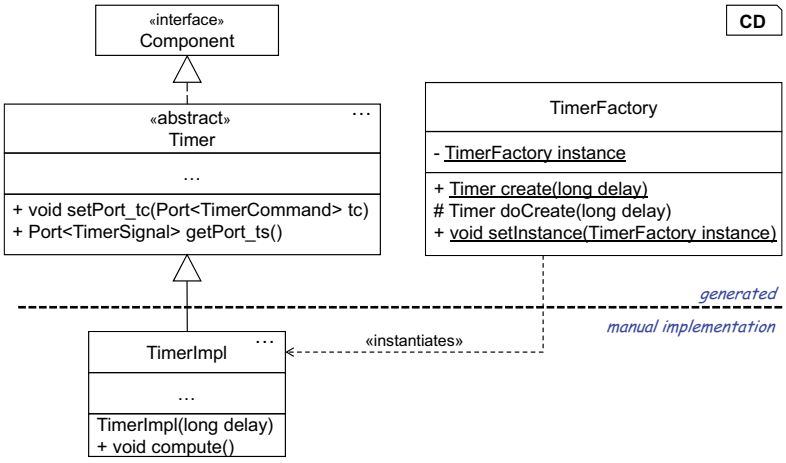


Figure 8.11.: The classes `Timer` and `TimerFactory` generated for the component parametrized `Timer[long delay]` from Listing 8.10 which requires a manual implementation supplied in the class `TimerImpl`.

the manual implementation is the name of the component with the suffix `Impl`. The example shown in Figure 8.11 also demonstrates how the Java code generator supports parametrized components. The parameter `delay`, required by the component, is added to the constructor of the component `TimerImpl` and to the method `create(long delay)` of the factory.

The manual implementation of a component has to extend the abstract class generated for the atomic MontiArcAutomaton component. The minimal manual implementation required for a MontiArcAutomaton component is the `compute()` method. The code generator already provides `setUp()`, `init()`, and `update()` methods that handle the instantiation and management of ports and variables. For parametrized components as the component `Timer[long delay]` a constructor has to be implemented that may call the constructor of the parent. An excerpt of the manually implemented `compute()` method of the class `TimerImpl` is shown in Listing 8.12.

Java

```
1  private long start;
2  private boolean set;
3  private TimerCmd lastCmd;
4
5  public void compute() {
6    if (tc.currentValue == TimerCmd.DELAY ||
7                          tc.currentValue == TimerCmd.DOUBLE_DELAY) {
8      start = System.currentTimeMillis();
9      set = true;
10     lastCmd = tc.currentValue;
11   }
12   else if (tc.currentValue == TimerCmd.ABORT) {
13     set = false;
14   }
15
16   if (set && ( (lastCmd == TimerCmd.DELAY &&
17                 System.currentTimeMillis() - start > delay) ||
18                 System.currentTimeMillis() - start > 2 * delay)) {
19     set = false;
20     ts.nextValue = TimerSignal.ALERT;
21   }
22 }
```

Listing 8.12: A manual implementation of the `compute()` method of the parametrized component `Timer[long delay]`.

Manually written code interacts with the API provided by our framework. In lines 6 and 7 of Listing 8.12 the manually written code reads the values received on the input ports from the attribute `tc` of type `Port<TimerCmd>` that represents the input port `tc` of the component `Timer`. Similarly, messages are sent by setting the attribute `nextValue` of the output ports. An example of sending the message `ALERT` on the output port `ts` of the `Timer` component is shown in line 20 of Listing 8.12. Manual implementations may use variables defined in the MontiArcAutomaton component type definition or introduce their own local variables as seen in lines 1-3 of Listing 8.12.

8.3.2. Code for Deployment

The MontiArcAutomaton Java code generator generates code for all component type definitions in the input path of the code generator. To deploy the code generated for a component to a Lego robot the component has to be marked with the stereotype «`deploy`». In the example of the bumper bot with the emergency feature shown in Figure 8.2 the top component `BumperBotEmergencyStop` is marked with the stereotype «`deploy`». The code generator generates the additional Java class `DeployBumperBotEmergencyStop` shown in Listing 8.13.

Java

```
1 public class DeployBumperBotEmergencyStop {
2   public static void main(String[] args) {
3     Component cmp = BumperBotEmergencyStopFactory.create();
4     cmp.setUp();
5     cmp.init();
6     while (true) {
7       cmp.compute();
8       cmp.update();
9     }
10  }
11 }
```

Listing 8.13: The code generated for the deployment of the component `BumperBotEmergencyStop` shown in Figure 8.2 on the LeJOS platform.

The generated class has a `main` method that calls the factory for the composed component `BumperBotEmergencyStop` to create an instance of the component. The code for deployment then executes the methods `setUp()` and `init()` of the component as shown in lines 4 and 5 of Listing 8.13. The deployment code starts an infinite loop iterating the compute and update cycles of all components.

The generated classes for deployment can be compiled and linked by the leJOS Java compiler and linker[1]. The leJOS compiler compiles Java files for the leJOS specific implementations of the standard Java libraries. The leJOS linker packages the compiled classes for the NXT brick. All classes required to execute a main class, e.g., the class `DeployBumperBotEmergencyStop` from Listing 8.13, are packaged in one archive that can be uploaded to the NXT brick.

8.3.3. Simulation Example

A robotics simulation environment allows the execution of the software of a robot without depending on physical hardware. It is essential for a simulation to execute the same code as deployed on the physical robot. To simulate and visualize the execution of the code generated by our code generator we have experimented with the SimBad simulator [HB06]. The SimBad simulator is written in Java and can execute Java code that controls a mobile robot in a simulated three-dimensional world.

A SimBad simulation requires a three-dimensional environment that can be created via SimBad APIs and an agent implemented in Java. An agent is an object in a three-dimensional world that can be equipped with sensors. Agents periodically read their sensors and can set their translational and rotational velocity. It is possible to simulate multiple agents at once [HB06].

[1] See leJOS compiler and linker documentation: `http://www.lejos.org/nxt/nxj/tutorial/Preliminaries/CompileAndRun.htm` (accessed 11/13).

To simulate the bumper bot from our example we have created a class that implements a SimBad agent with a touch sensor. The agent instantiates the component `BumperBot` as shown in Figure 8.14. For the instantiation of the component `BumperBot` it uses the factory generated by the Java code generator. All subcomponents of the component `BumperBot` are instantiated using their corresponding factories.

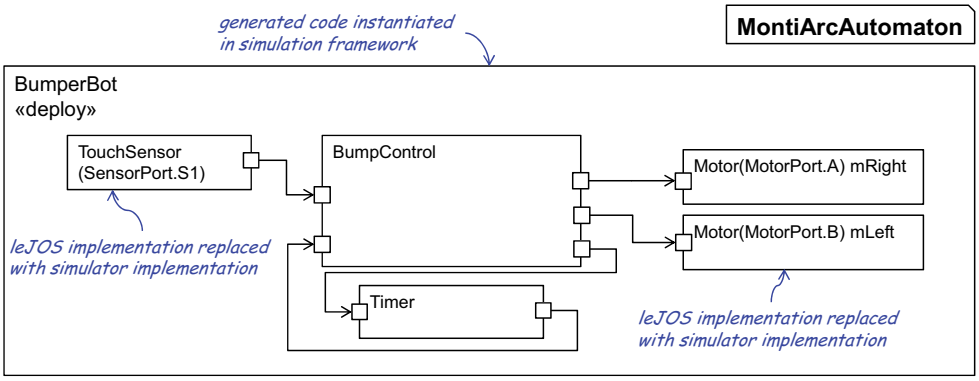


Figure 8.14.: The component `BumperBot` as instantiated in the SimBad simulator. The NXT platform specific implementations of the sensors and actuators are replaced by simulator specific implementations.

The factory mechanism allows the simulator to provide a custom factory for component instances of the type `TouchSensor` and `Motor`. During simulation the simulator periodically triggers the the agent that hosts the component `BumperBot`. The agent reads the sensor values provided by the simulation and translates them into signals on the bumper bot's touch sensor. The commands received by the simulation specific motor implementations are translated into the rotational and translational velocity of the robot. The agent then executes the `compute()` and `update()` methods of the `BumperBot` component instance. A visualization of the simulation is shown in Figure 8.15.

Our preliminary experiments with the SimBad simulator show that it is possible to execute the generated for the Lego NXT robots in a simulation environment without modifications. Necessary replacements of sensor and actuator component implementations are enabled by the factory pattern implemented in the generated code.

8.4. Case Study: Robotic Coffee Service

To evaluate the modeling language MontiArcAutomaton and our code generator, we held a one-semester course on model-driven robotics software development. From the first day of the course our students were working with the modeling language MontiArcAutomaton. The course was divided into three stages. During the first stage, a preparation

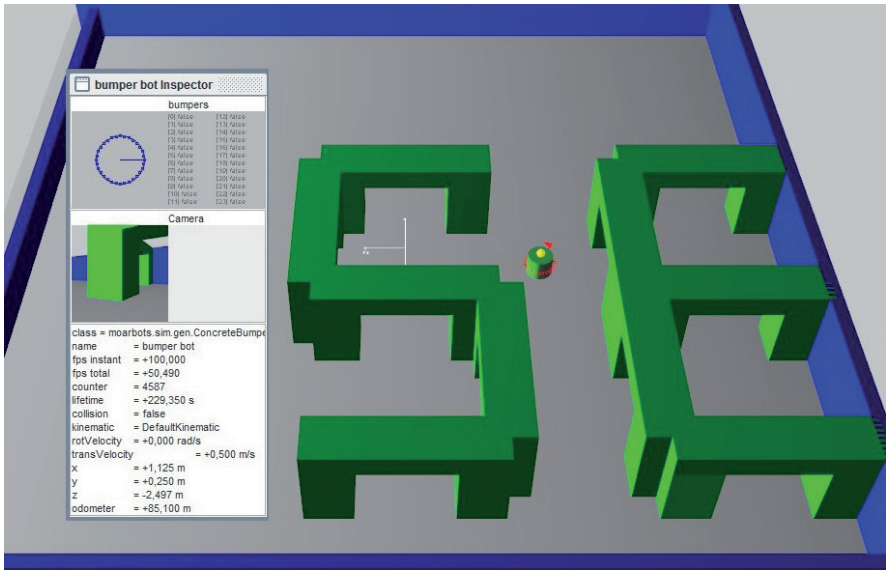


Figure 8.15.: Simulation of a single bumper bot using the generated Java code from MontiArcAutomaton models in the simulator SimBad [HB06].

phase before the start of the course, the students were assigned to prepare presentations about the modeling languages, tooling, and infrastructure based on available documentation. Aim of the second stage was to develop a robotic coffee service. In the third stage, the students improved the MontiArcAutomaton tooling based on the experiences from the second stage.

We conducted this course as a case study on the usage of model-driven engineering in the robotics domain in general and on the benefits of using the MontiArcAutomaton modeling language in particular. Using discussions, surveys and key figures from the students' development behavior, we tried to determine whether MontiArcAutomaton can be applied for the development of robotic software and which tools are most essential for successful model based development of robotic software.

Please note that we have only evaluated the modeling language MontiArcAutomaton and model-based development using the MontiArcAutomaton Java code generator. In this case study, the students have not applied the specification, synthesis, and analysis techniques presented in previous chapters.

Section 8.4.1 explains the course structure, background, and aims. Section 8.4.2 reports on the project results of stage two. Afterwards, Section 8.4.3 describes the results of our survey, discussions with the students, and key figures of their development behavior.

8.4.1. Project Description

During our course, the eight participating master level students learned model-driven engineering, development of robot control software, agile development using Scrum, and development of modeling tools. The students were evaluated weekly based on their reports and developed artifacts.

The course was divided into three stages: During stage one (nine weeks), the eight participants prepared presentations on technologies and practices, e.g., Scrum [SB01], JUnit [wwwi], MontiCore [KRV10], and MontiArcAutomaton, to be applied during development. In stage two (six weeks) the students developed a system of robots able to provide a coffee service using Lego Mindstorms robots. The robots development stage ended with a presentation of the working system[2]. Afterwards, we surveyed the students on their modeling experiences during the development stage. In the third state (ten weeks), the students were assigned to improve the MontiArcAutomaton tooling based on their answers from the survey and discussions.

During the latter two stages, the team was led by a student Scrum Master and a student Product Owner. We enacted the Scrum roles Customer and User to provide requirements and feature decisions. We participated in weekly sprint meetings to plan the next sprint and review the previous one. Due to the participants schedules, the development team was unable to have daily Scrum meetings. Weekly participation was mandatory to pass the class.

Goals

The goals of our study are to (1) determine whether MontiArcAutomaton can be applied for the development of robotic control software, (2) which tools are most essential for successful model-based development of robotic software. More specifically, we want to determine (1a) whether the decomposition of a robotic system can be adequately modeled using MontiArcAutomaton and whether (1b) the control logic and behavior of components can be adequately modeled using the I/O^{ω} automata paradigm [Rum96, RRW12] with time-synchronous communication. We also wanted to find out (2a) whether the existing tools for code generation and context condition checks were missing any features and (2b) which tools were missing for effective and efficient development of robotic software.

Students and background

All of the participating students had selected our course as first choice out of three choices. All but one student had previously attended the lecture *Software Engineering* with a basic introduction of model-based software development. Five of the students had attended the lecture *Model-Based Software Engineering* on UML and other modeling languages. Two of the students had attended the lecture *Generative Software Engineering* on DSL and code generator development using MontiCore. Three of the students had

[2]See the video at `http://www.se-rwth.de/materials/ioomega/#RoboticsLab`

experience with the Lego NXT platform. Two of the students are professional mathematical technical software engineers and have programming experience from industry jobs. One of the students conducted his bachelor thesis using the MontiCore code generation framework and the architecture description language MontiArc.

None of the students had experience with the modeling language MontiArcAutomaton and the runtime framework developed for the Lego NXT robots. In the first stage of the project each student was assigned a topic to prepare a presentation in front of the group. The topics covered in the presentations were: development tools (SVN, JUnit), leJOS, MontiCore (AST generation), MontiCore (language composition), MontiCore (code generation and templates), MontiArc, MontiArcAutomaton (language), and MontiArcAutomaton code generation. The slides of the talks presented by the participants of the case study are available from [wwwt].

Available material

For the first stage of the course we supplied our students with technical reports, papers, and presentations about the technology and tools to use. The materials included an introduction to the modeling language MontiArcAutomaton as presented in Chapter 6 and a draft of the MontiArcAutomaton technical report [RRW14].

For the second stage the students were supplied with two Lego NXT education kits and additional four regular Lego NXT sets. The NXTs were running Java using the leJOS firmware [wwwh]. We provided the code generator for MontiArcAutomaton models to Java code and a runtime environment with a library of platform specific components for the NXT leJOS platform described in Section 8.2. The students were using services of the SSELab [HKR12] that include a wiki, a bug tracking system, a mailing list, and an SVN repository to collaborate.

Task and user stories

The task of the students in the development stage (second stage) was to develop a system of autonomous robots that is able to receive coffee orders and deliver coffee to different offices and places. The initial task was described in 10 user stories:

1. As coffee drinkers, we want to instruct the robot to bring us fresh coffee.
2. As cleaning personnel, we do not want our cleaning cart to be knocked down by the robot.
3. As employees of the department, we do not want the robot to drive around meaningless in our offices.
4. As coffee drinkers, we want the robot to be able to fetch coffee with sugar.
5. As coffee drinkers, we want the robot to be able to fetch coffee with milk.
6. As robot owners, we do not want the robot to fall down stairs.
7. As impatient people, we want the robot to signal us, if it is waiting at a closed door.

8. As cleaning personnel, we want the robot to leave the coffee machine as clean as it found it.
9. As coffee machine owners, we want the robot not to operate the coffee machine when there is no water.
10. As agile scientists, we want the robot to deliver coffee to the following places: rooms 4304, 4312, 4315, and the sofa in the hall.

We gave no explicit orders on the number of robots to develop and many of the design and implementation decisions were left open, e.g., the implementation of ordering a coffee via a robot, web browser, or smart phone. Following Scrum practice the requirements were elicited, refined, and discussed in regular meetings.

Analysis procedure

Our analysis of the project is based on informal discussions with our students, results of a questionnaire, and statistics of their development behavior based on the version control system. In weekly meetings we had discussions about problems encountered during each previous sprint. We documented these discussions, which contained technical or organizational problems of the team, as informal notes. After the development stage and the presentation of the running system we discussed the deficiencies of the existing tools and the necessary improvements and additional tools to be created.

To capture the subjective evaluation of our students we created a questionnaire filled out after the development stage. The questionnaire contains 14 questions about the efforts spent on learning and development, about the confidence in the models and implementations, about the effort of fixing bugs and the amount of testing.

To complement the results of the survey, we also studied the students' development behavior. Therefore we identified several interesting key figures and monitored these from October 21 to November 17 at a three days sampling rate. The key figures identified are number of architecture changes in composed components, behavior changes in ${}^{*}\mathrm{MAA}_{ts}$ automata, and changes in Java files as well as the total number of composed and automaton components and number of components with Java implementations.

We furthermore counted for each project the number of components that were defined and instantiated within this project and the number of components from a library or runtime environment that were instantiated within this project in the final robot implementations.

8.4.2. Project Results

The students solved the second stage by implementing a system of three cooperating robots that receive a coffee order via a website, pick up a mug, fetch coffee, and deliver it to the person having placed the order. The system consists of a mug provider robot, a coffee preparation robot, and a coffee delivery robot as depicted in Figure 8.16. The coffee preparation robot is connected to an Android cellphone via Bluetooth, which hosts the coffee service website. After a user requests a coffee via the website, this issue is

forwarded to the coffee preparation robot, which informs the coffee delivery robot to pick up a cup, fetch coffee, and deliver it to the user.

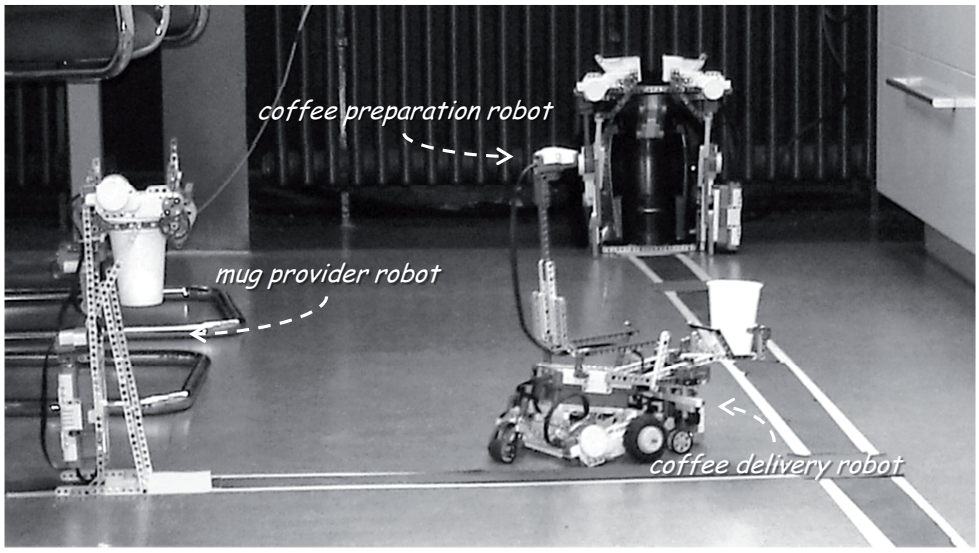


Figure 8.16.: The coffee service at work. A coffee request triggered the coffee service robot to pick up a mug and proceed to fetch coffee. A video is available from our website [wwwt].

Structure and behavior of all robots were modeled using MontiArcAutomaton. From these models, Java implementations were generated with the code generator we provided. The application models interface the robot using library components wrapping respective parts of the leJOS API.

The coffee delivery robot consists of ten component type definitions. Four of these contain automata and four others are composed. The remaining two components have Java implementations. The robot further reuses 15 components from the library, e.g., wrappers for hardware sensors and actuators.

Figure 8.17 shows the composed component `NavigationUnit` implementing the navigation of the robot. The component `NavigationController` for example determines the robot's next actions using an automaton. The inputs that the automaton reads come from several components wrapping access to the NXT's buttons `Button(1)`, `Button(2)`, and `Button(3)`, and augmented readings from two color sensors (component instances `csl` and `csr` of type `ColorSensor`). The automaton inside the component `NavigationController` has eleven states and 41 transitions. The component `DifferentialPilot` is implemented in Java and uses a part of the leJOS framework to move the robot.

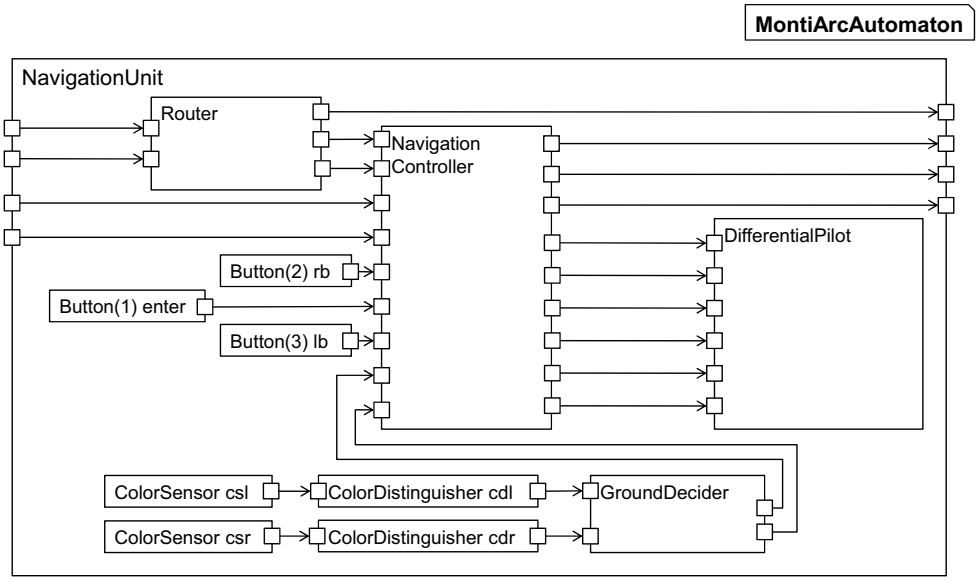


Figure 8.17.: Structure of the composed component `NavigationUnit`.

8.4.3. Analysis and Interpretation

Our analysis covers results from weekly discussions with all students, model changes from version control, and a survey.

Results from weekly discussions

In weekly meetings, we conducted regular discussions with the students about the issues at hand. During the development stage, the students had problems structuring the development tasks which yielded claims for better communication and for a dedicated system architect administering interface changes of MontiArcAutomaton components. Due to the lack of arrangement, the Bluetooth communication was implemented three times: first, as a detailed design document, second, as a deviating implementation, communicating complex data structures not suitable for MontiArcAutomaton, and third, as another deviating implementation, communicating enumeration types.

Similarly, the initially developed and agreed upon architecture of the team was dismissed unused. Another important issue was the lack of tool support to facilitate development. The students claimed that editor support (e.g., model completion, context conditions) is crucial to efficient model-driven engineering and thus created a text editor with these features in the third project stage. Other technological issues expressed were the parallel development of both software and hardware and the manual deployment of components to platforms.

Regarding MontiArcAutomaton, our students had initial problems to restrict component communication to simple message passing using automata. On the other hand, they did not miss language features like hierarchical states in MontiArcAutomaton automata. This supposedly is due to the existence of composed components, which allow similar decomposition mechanisms using subcomponents. Please note that seven students attended the lecture *Software Engineering* teaching basic concepts of Statecharts while five students in addition attended the lecture *Model-Based Software Engineering* with exercises on more complex features, e.g., history states or entry actions. We are not aware of Statechart modeling experience of the students beyond these lectures.

On a less technical note, the students mentioned that Scrum may be suboptimal when the semester schedule prohibits daily Scrum meetings. Unfortunately, this issue was beyond our control. While the discussions mostly addressed process issues, the questionnaire focused on the effort required to model robot control software using MontiArcAutomaton.

Results from the questionnaire

After the students finished the development stage with a presentation of the running system, we conducted a survey using the questionnaire available on the MontiArcAutomaton website [wwwt]. We asked the students 14 questions on their development efforts, problems, confidence in the artifacts, and testing behavior. We handed out eight questionnaires and all eight were returned fully filled.

The results suggest that learning MontiArcAutomaton was a major effort during the development stage of the course and that this effort can be eased by helpful development tooling (e.g., editor support with early feedback). The students reported to have implemented only two regression tests, but many manual tests of their systems.

First, we asked the students how much time they have spend learning our technologies. The results, displayed in Figure 8.18 (a), point out that learning the leJOS robot API was significantly easier than learning the modeling language MontiArcAutomaton. We believe this is partly due to the amount of available well-written documentation for leJOS, many examples, and an active online community. Compared to this, information on MontiArc and MontiArcAutomaton was only available from two technical reports, one paper, and about 30 example models.

The second question of the questionnaire asked what percentage of the time the students spent on using MontiArcAutomaton for modeling the structure of the system, using MontiArcAutomaton for modeling component behavior, using Java for programming component behavior in leJOS, and how much time they spent on the construction of the robots. Figure 8.18 (b) shows, that the students spent most of their time modeling the behavior of components, followed by the time spend to construct the actual Lego robots. While the former did not surprise much, we learned from observations that it was surprisingly hard for some students to construct useful Lego robots without building instructions.

We asked the students to estimated the effort required to understand composed component models, automaton models, and Java implementations on a scale from 1 (simple)

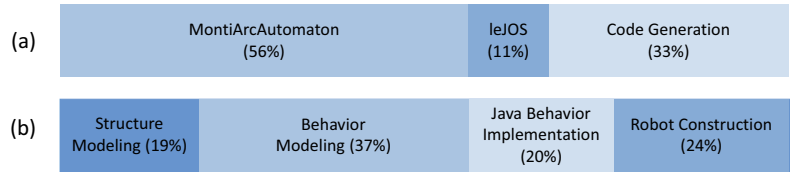


Figure 8.18.: Fractions of the time spent on (a) learning the technologies and on (b) creation of the three robots of the coffee system.

to 10 (almost impossible). The results displayed in Figure 8.19 (a) show that the students found composed components as comprehensible as Java, which they had been introduced to in their first semester. Automata were considered harder to understand.

While we expected similar results, we were surprised by the students' feedback that the effort for fixing bugs[3] in composed component type definitions was assumed to be as high as the effort for fixing bugs in automata (Figure 8.19 (b)). The students considered the effort for fixing bugs in MontiArcAutomaton twice as high as the effort for fixing bugs in Java artifacts. Discussions yielded the result that this is due to Java being taught from the first semester and due to the lack of development tooling for MontiArcAutomaton such as a debugger.

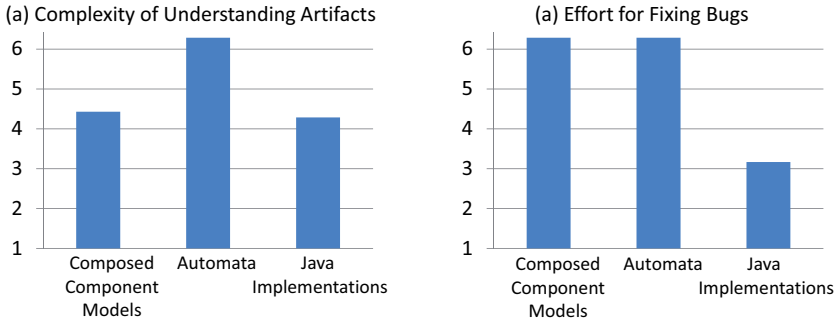


Figure 8.19.: Efforts for understanding the different development artifacts and fixing bugs as rated by the students on a scale from 1 (simple) to 10 (almost impossible).

We also asked the students to estimate their confidence in the artifacts created by them and their team members on a scale from 1 (no confidence) to 10 (works perfectly). The answers are shown in Figure 8.20. While we found little difference between the confidence in the artifacts created by students themselves or others, the students were overall less confident in the automata than in the component and Java artifacts, which follows from the assumed complexity of automata.

[3]Our broad definition of *bug* covers any incorrect or unexpected behavior of the software (sub-)system.

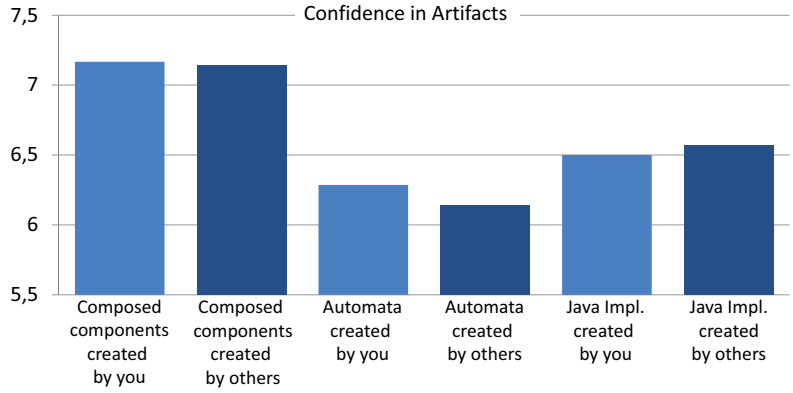


Figure 8.20.: Confidence in the correctness of different development artifacts as rated by the students on a scale from 1 (no confidence) to 10 (works perfectly).

The students claimed to have used automata for 57.3% of the atomic components and that they could have used automata to implement the behavior of up to 64.6% of the components. Actually they modeled 12 of the 23 (= 52.2%) components with automata and developed two components with Java implementations. If these would have been modeled with automata too, they would have modeled the total of 60.1% components with automata, which is close to their estimate. In conclusion, the students estimated, that they could have modeled all behavior implementations with automata. Overall, the results showed that learning a new modeling language poses the expected challenges and is even harder if lacking tool support.

Results from the students' development behavior

To complement the results of the questionnaire, we also monitored the students' development behavior as described in Section 8.4.1. We present several key figures.

For the three robot projects we have counted the numbers of MontiArcAutomaton components implemented as pure models (automata and composed components) and the number of components implemented in Java over time as shown in Figure 8.21. The rise after six days was an initial version of the robots created by the students mainly in Java. Another week later the components implemented as composed and automaton components started to outgrow the Java implementations. Of the two remaining Java implementations one is a value lookup component and the other one implements an obstacle detection and classification. A review of the code suggests that both could be implemented using automata with some additional effort. The development over time indicates the learning curve of our students. Initially, they created the implementation in Java and then switched to modeling all but two components using MontiArcAutomaton.

We have analyzed the changes of artifacts in the version control history of the students' projects. We distinguish between the library project with hardware wrappers

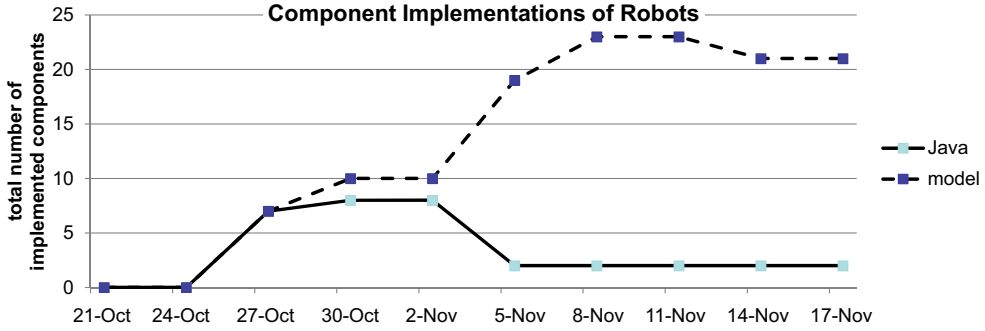


Figure 8.21.: The total number of components implemented as Java implementations and models.

implemented in Java and the three projects containing the robot control software.

The numbers of changes per artifact in the three robot projects are shown in Figure 8.22. Again we can see a difference from the beginning of development — with equal or more changes per Java file — to the second part of development with many more changes per model file. We see a peak of 2.4 changes per MontiArcAutomaton model around November 8 before the scheduled *final* presentation of the robots. On average and over the complete development time each MontiArcAutomaton model has been changed 8.6 times and each Java file 4.8 times.

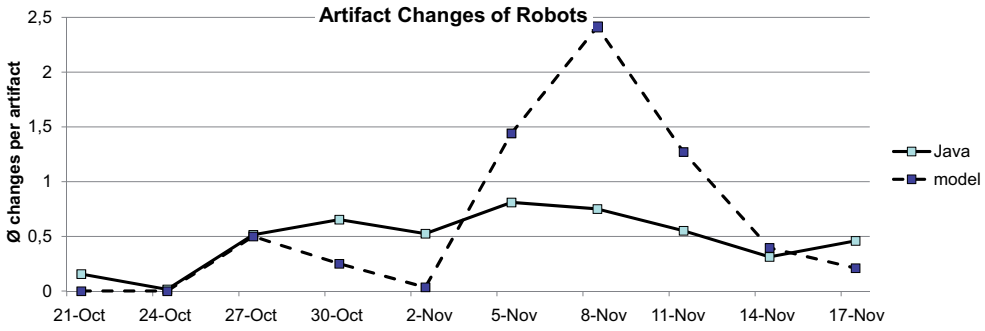


Figure 8.22.: The changes over time per Java and model file of the implementations of all three robots.

An interesting observation on the artifact changes in the library project is that models were changed on average 0.3 times while each Java file was changed 4.5 times. This indicates that component interfaces did not change much while component implementations did.

8.4.4. Threats to Validity

We performed this study on a single software project with a group of eight graduate students. The students took part in this course to obtain a certificate and data was gathered using discussions, a survey, and development figures. This setup yields threats to the internal validity (causality) and to the external validity (generalizability) of this study.

Threats to the internal validity stem from the students' lack of previous experience using MontiArcAutomaton. They had to learn and apply the MontiArcAutomaton language before they could determine any benefits for modeling robotics software. Another study, wherein the same group of students develops a new robot control software with prior knowledge of MontiArcAutomaton could resolve this issue. Further threats to causality follow from the instruments we chose to determine the students opinion on modeling: Questionnaires are subject to several issues, e.g., the scales are understood differently by participants, there are several well-known response biases, and the results only reflect the participants self-perception.

Threats to generalizability ensue from the number of participating students and the fact that the students were graded. While the first threat can be eliminated by future experiments with a greater number of students, omitting grades is not feasible in the setting of a university class.

8.5. Discussion

This section presents a discussion of the results of the case study and several advanced features regarding MontiArcAutomaton code generation. We discuss the language elements supported by code generation and the relation between the code generation and the analysis framework presented in Chapter 7. We also discuss future work to support components that are only weakly and not strongly causal, and we discuss challenges in component distribution.

8.5.1. Case Study Discussion

Modeling robot control software poses several challenges. In both discussions and survey, the students pointed out that learning the ${}^{*}\text{MAA}_{ts}$ automaton part of MontiArcAutomaton was rather hard. On one hand one could expect that ${}^{*}\text{MAA}_{ts}$ automata with their similarity to Mealy machines or UML Statecharts are easy to learn, on the other hand, mastering a new language and paradigm for application development is never easy. This problem was amplified due to the lack of tooling supporting the modeling process. Despite these issues, the students developed 86% of the atomic components using MontiArcAutomaton and believe that they could have modeled the remaining atomic components also as automata instead of providing manual Java implementations.

MontiArcAutomaton is implemented as a MontiCore language, which allows to embed arbitrary MontiCore behavior modeling languages into components. Thus, specific modeling languages for common robotics problems may further ease robotics software

development. We have reported on our plans for extending the modeling language MontiArcAutomaton with robotics domain-specific languages in [RRW13c].

During the development stage, the students especially claimed that editor support is crucial to efficient model-driven engineering. They therefore developed a text editor with several features and integrated it into a graphical editor Eclipse plug-in for $^*\text{MAA}_{ts}$ automata[4]. To facilitate deployment, they further developed a deployment language profile of MontiArcAutomaton, which maps components to platforms and communication technologies.

8.5.2. Supported Elements of MontiArcAutomaton and $^*\text{MAA}_{ts}$ Automata

The MontiArcAutomaton Java code generator supports all features supported by the translation of MontiArcAutomaton into Mona presented in Chapter 7. This includes component type definitions from Definition 6.8 and $^*\text{MAA}_{ts}$ automata from Definition 6.20 without the restrictions of the translation into Mona. The subset of $^*\text{MAA}_{ts}$ automata from Definition 6.20 supported by our translation into Mona is fully supported by the MontiArcAutomaton Java code generator. In addition, our code generator supports OCL/P guard predicates. The code generator also supports using types defined in UML/P class diagrams and primitive types, such as, `float`, `double`, and `int`.

Additional features of the MontiArcAutomaton modeling language that are not contained in Definition 6.8 of component type definitions are generic component types and parametrized component types. The MontiArcAutomaton Java code generator supports these language features by generating implementations that use similar concepts of the Java programming language. Generic component types are translated into generic Java classes and parametrized component types are translated into classes with factories that require values for all parameters to construct component instances.

8.5.3. Verification and Code Generation

The MAA_{ts} language profile of MontiArcAutomaton has been developed in parallel with the verification environment described in Chapter 7. After an initial MontiArcAutomaton verification prototype presented in [Kir11] we have started with the development of the MontiArcAutomaton Java code generator. The code generator implements the time-synchronous model of computation of MontiArcAutomaton components used in Chapter 7. We consider it an important feature of our work to support the same semantics for verification and code generation.

In early experiments with a prototype of the code generator used in the case study we found out that some features were missing from the language profile of MontiArcAutomaton that we had initially considered. Specifically, the preliminary language used in [Kir11] did not have support for $\varepsilon_{\downarrow}$ values and $\varepsilon_{\downarrow}$ completion. This feature was added to the modeling language and both the analysis and code generation prototypes. Other missing features were local variables and guard predicates on transitions. We have added

[4]See the video at `http://www.se-rwth.de/materials/ioomega/#Editor`

these features to the language profile as described in Chapter 6. The verification tool prototype presented in Chapter 7 also supports local variables but not guard expressions.

In the current tool implementations, all of the features of the verification tool are supported by the Java code generator. For the case study, the students were not using the verification tool. The implementation of the robotic coffee service employs the following features not supported by the verification prototype: ports of the Java types `Integer` and `Double`, OCL/P guard predicates, e.g., for comparing distances measured by ultra sonic sensors of the robots, and parametrized components, e.g., to provide initialization parameters for the speed of motors.

8.5.4. Strong vs. Weak Causality and Scheduling

The MontiArcAutomaton code generator generates code for components that are strongly causal. Strong causality is enforced by a computation and a separate update phase of all components deployed on one robot. Outputs of the computation phase are only available to other components after the global update phase. The students participating in the case study have not reported any difficulties with the delays introduced by strong causality.

We have discussed an extension of the MontiArcAutomaton semantics to also support weakly causal immediate processing of messages in Section 7.7.6. The composition of only weakly causal components may lead to contradictions and unrealizable behavior. The problem does not occur if every directed feedback cycle in a composition contains at least one strongly causal component.

In this case, the communication dependencies between atomic components, limited to one time slice, do not contain cycles. Thus, it is possible to compute a fixed scheduling of the atomic components and adapt our code generator to support weakly causal components.

8.5.5. Component Distribution

In many cases component and connector architectures are not only logically decomposed but also physically distributed to different processing units. The system developed in the case study described in Section 8.4 consists of four interacting devices: a cell phone to place coffee orders, a robot operating a coffee machine, a robot operating a cup dispenser, and a mobile robot that delivers coffee. The cell phone and the three robots communicate via Bluetooth connections.

The top level component type definitions for the deployment of the coffee preparing robot and the cup dispenser robot are shown in Figure 8.23. The two robots communicate via a their subcomponents of type `PrepCommunication` and `Communication`. These components wrap access to the Bluetooth interfaces of the respective NXT bricks. A message sent by the component `CommandController` of the coffee preparing robot is received by the component `PrepCommunication` and sent to the cup dispenser robot via Bluetooth. The component `Communication` of the cup dispenser queries the

Bluetooth interface in every execution cycle. Once it receives the messages it forwards it on a corresponding output port to the component `CupDispenserControl`.

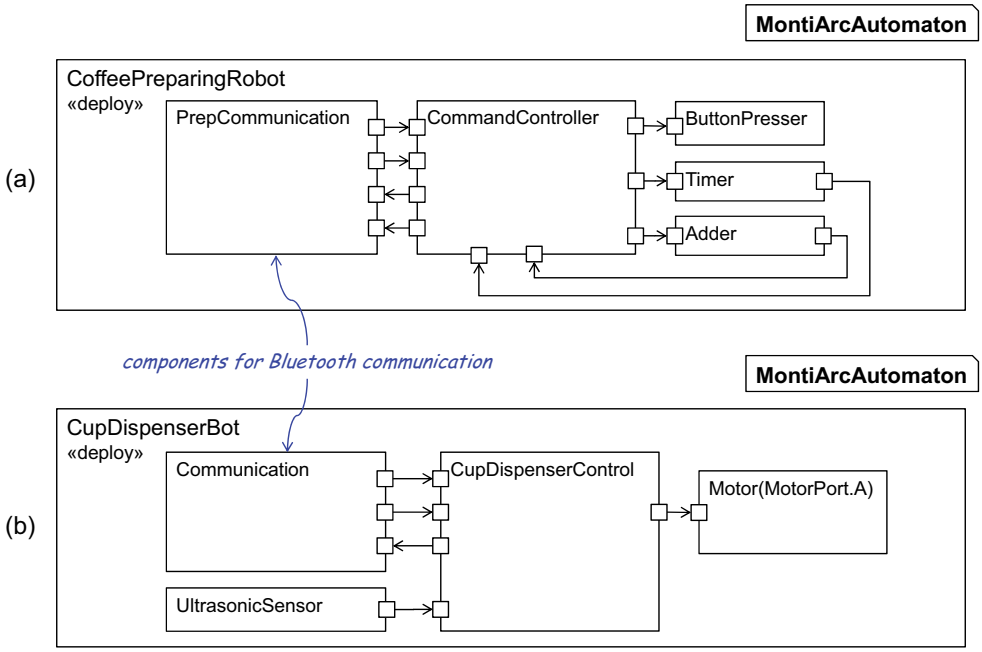


Figure 8.23.: The top level decomposition of (a) the coffee preparing robot and (b) the cup dispenser robot. The interaction between the robots is realized via the components `PrepCommunication` and `Communication` that provide access to the Bluetooth interface of the NXT brick.

The communication between the robots is not modeled in a way accessible for formal analysis. The Bluetooth communication interface is used similarly to an actuator that manipulates the environment and a sensor that observes it.

As a future work we are investigating methods that make the communication explicit by adding a special type of connectors representing Bluetooth communication between ports of the deployed components. This will include a synchronization of the currently independent computation and update cycles of the interacting robots.

8.5.6. Different Robotics Target Platforms

The code generation approach we have implemented is not limited to the educational Lego Mindstorms NXT robotics platform. We have also developed a code generator for industrial robotics platforms running the robot operating system (ROS) [QGC+09]. The code generation for ROS generates Python code.

The development of code generators for additional target platforms and different programming languages is facilitated by the MontiCore code generation framework. Many of the calculators we have developed for code generation are target agnostic. The Java and the Python code generation have seven calculators in common and each only two specific calculators [RRW13b].

We have reported on code generation from MontiArcAutomaton models to different target platforms in [RRW13b]. The main challenges in supporting a new target platform are (1) the development of an execution and scheduling mechanism that implements the time-synchronous semantics and (2) the manual implementation of hardware specific components using the native APIs of the target platform.

8.6. Related Work

The main goal of our code generator is to provide a model-based solution for the development of interactive C&C systems. An important factor for a continuous development process is the conformance of the execution of the generated code to the semantics of the modeling language MontiArcAutomaton. This ensures that analysis results established in previous development steps carry over to generated system implementations.

We discuss related code generators for state-based behavior modeling languages and formalizations of the modeling languages' semantics.

Many code generators are available for UML Statecharts [Obj12a]. Examples of code generators that produce Java code are Rational Rhapsody [wwwg] and Visual Paradigm [wwwab]. Ab Rahim and Whittle [RW10] have analyzed the conformance of UML Statechart code generators to the semantics described in the UML specification [Obj12a]. They have found that the Java code generated by Rational Rhapsody and Visual Paradigm deviates from UML semantics. It might be possible to apply Ab Rahim and Whittle's approach to our code generator and validated the conformance of the code generated from MontiArcAutomaton models.

Another modeling formalism related to MontiArcAutomaton is Matlab Simulink extended with Stateflow charts [wwwo]. A formal definition of the Stateflow semantics semantics is, e.g., given in [HR07]. Matlab Simulink provides a block library to model components of Lego NXT robots and a C code generator for the deployment and execution of Simulink models on Lego NXT robots [wwwk]. Similar to our code generator, Simulink supports manual implementations of blocks in block diagrams. It is thus possible to provide C code for direct access to the Lego NXT's APIs.

AutoFOCUS [BHS99, HF07] is a complete IDE with many graphical modeling languages for specification, implementation, testing, and deployment of reactive distributed systems. AutoFOCUS' semantics for automata is similar to the semantics of ${}^{*}\mathrm{MAA}_{ts}$ automata implemented in our MontiArcAutomaton Java code generator. Code generation from AutoFOCUS models is available for multiple target languages including C and Java [HS97, HST10]. We are not aware of code generators from AutoFOCUS models to robotic platforms or reports on case studies similar to ours.

Finally, MontiArcAutomaton is implemented as a MontiCore language, thus it fa-

cilitates the embedding other behavior modeling languages and integration of other MontiCore code generators [RRW13b, RRW13c]. These possibilities make MontiArc-Automaton and extensible framework for the development of interactive C&C systems.

Chapter 9.

Summary and Conclusion

In this thesis, we have developed description techniques that allow crosscutting model-based specifications of the structure and behavior of C&C systems. To enable efficient and effective system design, we have presented analysis and synthesis methods supporting model-based development of interactive C&C systems.

In the next section, we will briefly summarize the main results of our work. We discuss some of the limitations of the current results in Section 9.2 and suggest future work based on our evaluation in Section 9.3. We conclude our contribution in Section 9.4.

9.1. Main Results

The goal of this thesis was to support the development and evolution of interactive C&C systems. Our approach is based on domain-specific notations for modeling and specifying the structure and behavior of C&C systems at different levels of abstraction. We have introduced these notations to provide novel, model-based analysis and synthesis methods that guarantee the correctness of implementations.

We specifically focused on a modeling language for partial knowledge of the structure of C&C models and on a language for state-based component behavior modeling. Both modeling languages have powerful mechanisms for abstraction and underspecification. The specification mechanisms of these languages enable new approaches to documenting, analyzing, and synthesizing interactive C&C systems. We have formally defined the respective analysis problems and provided prototype implementations that serve as demonstrations for the feasibility of the new approaches.

Chapter 3 introduced a modeling language for C&C views. C&C views allow the documentation of partial knowledge of the decomposition and connectivity of a C&C system. C&C views offer powerful abstractions of ports, component hierarchy, and connectors. A variability mechanism within the views language allows extensions of the syntax and semantics of C&C views increasing the expressiveness of the language. The corresponding verification problem, investigated in Chapter 4, is to determine whether a given C&C model satisfies a C&C view. We have developed a polynomial verification algorithm that determines satisfaction and computes witnesses that demonstrate the reasons for verification results. The automated verification enables the development and evolution of C&C models that satisfy crosscutting structural constraints documented in C&C views. The generated witnesses support engineers in understanding verification

results.

A C&C views specification is a propositional formula over views that specifies valid, invalid, alternative, and dependent designs. We have shown in Chapter 5 that the problem of synthesizing a C&C model that satisfies a C&C views specification is NP-hard. We solve it by a reduction to SAT via Alloy. Our prototype tool enables the synthesis of C&C models from a set of crosscutting structural specifications that are correct by construction. We have developed a synthesis implementation using a translation into Alloy. Advanced features supported by our implementation are language extensions for atomic and interface-complete components, library components, and support for architectural styles. This shows that synthesis from C&C views is not only feasible, but also extensible to advanced features of architectural modeling.

We have presented the modeling language MontiArcAutomaton and a language profile for ${}^{*}\mathrm{MAA}_{ts}$ automata in Chapter 6. These automata provide underspecification mechanisms for specifying behavior of components and component compositions. Chapter 7 presented an analysis framework for component behavior based on a representation of MontiArcAutomaton semantics in Mona. A specification language allows to define MontiArcAutomaton specification checks that include behavior refinement and equality of components. Our verification environment provides engineers with means to analyze component behavior and verify behavior refinement from early specifications to component implementations.

Finally, we have developed a code generator for a robotics platform that supports code generation from MontiArcAutomaton models. Our code generator allows engineers to directly deploy modeled interactive C&C systems on robotic hardware or in simulation environments. It supports the integration of manually implemented components to access a platform's native APIs. The MontiArcAutomaton modeling language and our code generator were evaluated in a case study conducted with graduate students. The students developed a distributed robotic coffee service.

Our work provides modeling languages and prototype tool implementations supporting the development of interactive C&C systems during various phases of system development. This is achieved by providing mechanisms for expressing partial knowledge and crosscutting concerns regarding the structure and behavior of these systems.

In a larger scope, this work is an example for the rigorous development of model-based synthesis and analysis tools. The development of the four prototypes for C&C views verification, C&C views synthesis, MontiArcAutomaton component behavior verification, and MontiArcAutomaton code generation has followed a structured approach:

1. identifying relevant structures and modeling languages of implementations,
2. developing suitable modeling languages to express specifications,
3. relating the semantics of the investigated models,
4. defining the analysis or synthesis problem,
5. developing a translation or algorithm to solve the problem, and
6. translating the results back into the problem domain.

	single comp. inst.	multiple comp. inst.	generic comp. types	param. comp. types	port types	connectors	library/manual impl.
Language: C&C models (Ch. 2)	✓	-	-	-	✓	✓	✓
Language: C&C views (Ch. 3)	✓	-	-	-	✓	✓	✓
Application: C&C views verification (Ch. 4)	✓	-	-	-	✓	✓	✓
Application: C&C views synthesis (Ch. 5)	✓	-	-	-	✓	✓	✓
Language: MontiArcAutomaton (Ch. 6)	✓	✓	✓	✓	✓	✓	✓
Application: MAA_{ts} analysis (Ch. 7)	✓	✓	-	-	✓[a]	✓	✓
Application: MAA_{ts} code generation (Ch. 8)	✓	✓	✓	✓	✓	✓	✓

[a]The port types supported by MAA_{ts} verification are UML/P enumerations and other finite types.

Table 9.1.: Overview of the concepts for modeling C&C systems supported (✓) or not supported (-) by the languages and applications presented in this thesis.

9.2. Limitations

We have examined some limitations of the introduced languages and prototypes in dedicated discussion sections of the chapters throughout this thesis. In this section, we discuss limitations of our current work in the context of a system development process based on the presented techniques. An integrated development process would ideally combine all presented approaches from initial structural and behavioral specifications to executable code. We highlight differences between the developed languages and techniques that currently impose limitations on such a process.

The modeling languages and description techniques presented in this thesis differ in the concepts for modeling C&C software architectures they support. The main language concepts of C&C systems that we use for a comparison of the presented techniques are component instantiation, component types, port types, connectors, and support for library and manually implemented components. An overview of these concepts and their support by the modeling languages and applications introduced in Chapters 2-8 is given in Table 9.1.

The modeling language for C&C views presented in Chapter 3 is a specification language for the structure of C&C models introduced in Chapter 2. C&C models represent instances of C&C systems as hierarchically decomposed components with well-defined interfaces communicating via connectors between the ports of components. The C&C views verification problem and the C&C views synthesis problem are defined for C&C views and C&C models. The developed techniques focus on component instances and not

on component types and thus do naturally neither provide an instantiation mechanism nor support generic and parametrized component types as shown in Table 9.1.

The modeling language MontiArcAutomaton introduced in Chapter 6 supports component types and their instantiation. It also supports multiple instantiation of components, generic component types, and parametrized component types. While all these features are supported by the MontiArcAutomaton code generator introduced in Chapter 8, generic and parametrized component types are not supported by the Mona verification presented in Chapter 7, as indicated in Table 9.1 and discussed in Section 7.7.2.

The differences in the C&C modeling concepts supported by the techniques presented in this thesis constrain a seamless integration. The set of supported concepts is nonetheless increasing from C&C views verification to MontiArcAutomaton code generation. This does allow transitions from modeling the structure of C&C models to behavior implementations using MontiArcAutomaton and to code executed on robots.

9.3. Recommendation for Future Research

In previous chapters we have suggested future work and experiments to extend and evaluate the applicability of the developed techniques and to improve performance of the analyses. We have suggested future research motivated by the results of our evaluation and by related work in the discussion sections of each chapter.

One future work is extending the language for C&C views with additional abstractions, e.g., of component names, and with quantification over components and ports (see Section 4.4.4). These extensions might be implemented using the existing variability mechanism of C&C views. We consider it very important to balance the expressiveness of the language with its intuitiveness, i.e., the 'by example' nature of C&C views.

An important future work on C&C views synthesis is handling unsatisfiable specifications. The attempt to synthesize a C&C model for a C&C views specification might have different reasons to fail. One reason is the limit of the manually chosen synthesis scope. A satisfying C&C model might exist in a larger scope. Another reason is that the C&C views synthesis specification might be unsatisfiable in any scope. As a future work we propose to analyze possible reasons for the unsatisfiability of a given specification and report these to the user (see Section 5.7.7).

We have presented a verification framework for the behavior of MontiArcAutomaton components. We evaluated the supported refinement and equality checking on example systems. The results of running times clearly show a limitation of the framework for verifying larger models (see Section 7.7.1). This limitation is due to our current translation into WS1S formulas solved by Mona. A future work regarding MontiArcAutomaton component verification is either an improved translation into WS1S or an implementation of the verification based on other techniques and model checkers.

Our code generator for MontiArcAutomaton models generates code that can be directly executed on Lego NXT robots. The generated code is executed in cycles implementing the time-synchronous semantics of MontiArcAutomaton. The code generator does currently not support the scheduling of code on physically distributed systems. As a

future work we suggest to model the deployment of components to physically distributed systems and support the synchronization of these systems to ensure the time-synchronous semantics (see Section 8.5.5).

Apart from the improvement of the proposed techniques and prototypes, their integration also poses interesting research questions. We have identified some limitations in the applicability of the development techniques in an integrated process in Section 9.2. These limitations suggest extending C&C views and the C&C views satisfaction relation to component type definitions. We discussed this extension in Section 3.7.3. Additionally, the integration of the methods requires an extension of the MontiArcAutomaton verification framework to handle all concepts supported by MontiArcAutomaton code generation.

Important phases of the life-cycle of interactive C&C systems are maintenance and evolution [MMR10]. These phases are currently partially covered, e.g., by the support of regression testing using C&C views and the verification of MontiArcAutomaton component behavior. Our current implementations mostly handle structure and behavior independently while they are often not. For example, the C&C views synthesis process does not consider component behavior. Changes in the structure of components, however, might influence the behavior implementations in MontiArcAutomaton models.

We thus suggest investigating theories and tools for co-evolution of the structure and behavior of C&C systems. FOCUS [Bro93, BS01] provides necessary theories for these tasks and existing works already address handling structural and behavioral changes that preserve or refine behavior [PR99, Gru05]. We propose the integration and automation of these techniques in our model-based development framework as future work.

9.4. Conclusion

We presented novel modeling languages for the specification of the structure and behavior of interactive C&C systems.

C&C views specify structural properties of C&C models in an expressive and intuitive way. C&C views provide means to abstract away direct hierarchy, direct connectivity, port names and types, and thus can crosscut the traditional boundaries of the implementation-oriented hierarchical decomposition of systems and subsystems. MontiArcAutomaton is a modeling language for modeling component composition and state-based component behavior. The language provides powerful underspecification mechanisms and MontiArcAutomaton models can be used as specifications as well as implementations of component behavior.

We have presented model-based analysis techniques to verify the structure of C&C models using C&C views and to check refinement and equality between MontiArcAutomaton component behavior definitions. The verification prototypes translate the verification results back to the problem domain and provide witnesses that demonstrate reasons for non-satisfaction.

We implemented a synthesis prototype that given a C&C views specification consisting of mandatory, alternative, and negative views, computes a C&C model that satisfies the

specification, if one exists. The result of synthesis can be used for further exploration or as the complete, final model itself. For MontiArcAutomaton models we have developed a code generator that generates executable code for a robotics platform.

The prototypes that solve these novel analysis problems were evaluated on example systems, in an online user study, and in a case study of a distributed robotic system. Our evaluation shows the feasibility of the analysis and synthesis tasks and the applicability of the modeling languages for developing interactive C&C systems.

Our results go beyond the applications introduced in this thesis. The generic implementation of C&C models in Alloy presented in Chapter 5 allows arbitrary analyses of C&C models. Similarly, our translation of C&C views into Alloy is not limited to the current language features of C&C views. We have demonstrated the translation's extensibility and its support for the language variability mechanism of C&C views in Section 5.4.1. Thus, we provide a general and extensible solution for formal C&C model analyses.

The translation of MontiArcAutomaton components into Mona, which we have presented in Chapter 7, is not specific to checking refinement and equality. The resulting predicates express finite I/O relation semantics of MontiArcAutomaton components as Mona predicates. Additional analyses may employ this translation, e.g., to check whether a component satisfies incomplete observations of messages defined in sequence diagrams.

Finally, the modeling language MontiArcAutomaton is the basis of an extensible robotics modeling framework with support for multiple robotics platforms and target programming languages. The extensibility of the language allows embedding modeling languages other than automata to define component behavior. MontiArcAutomaton constitutes a powerful framework beyond the results presented in this thesis.

We believe that our work provides promising results and suggest interesting future research directions towards a comprehensive model-based development environment for interactive component and connector systems.

Bibliography

[AAAN+08] Marwan Abi-Antoun, Jonathan Aldrich, Nagi H. Nahas, Bradley R. Schmerl, and David Garlan. Differencing and merging of architectural views. *Autom. Softw. Eng.*, 15(1):35–74, 2008.

[ABG+13] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[BCK+11] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *Comput. J.*, 54(5):754–775, 2011.

[BCL12] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. A systematic review of software architecture evolution research. *Information & Software Technology*, 54(1):16–40, 2012.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} States and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TUM-I9202, SFB-Bericht Nr. 342/2-2/92 A, 1992.

[Bee94] Michael von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[Bie97] Armin Biere. μcke - Efficient μ-Calculus Model Checking. In *CAV*, pages 468–471, 1997.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKGS11] Ajinkya Bhave, Bruce H. Krogh, David Garlan, and Bradley R. Schmerl. View consistency in architectures for cyber-physical systems. In *ICCPS*, pages 151–160. IEEE, 2011.

[BL94] Manfred Broy and Leslie Lamport. The RPC-Memory Specification Problem - Problem Statement. In Manfred Broy, Stephan Merz, and Katharina Spies, editors, *Formal Systems Specification*, volume 1169 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 1994.

[BL01] Benedikt Bollig and Martin Leucker. Modelling, specifying, and verifying message passing systems. In *TIME*, pages 240–247. IEEE Computer Society, 2001.

[BL06] Benedikt Bollig and Martin Leucker. Message-passing automata are expressively equivalent to emso logic. *Theor. Comput. Sci.*, 358(2-3):150–172, 2006.

[Box98] Don Box. *Essential COM: The Component Object Model*. [DevelopMentor Series. Addison-Wesley Professional, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007.

[Bro93] Manfred Broy. (Inter-)Action Refinement: The Easy Way. In Manfred Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*. Springer NATO ASI Series, 1993.

[Bro97a] Manfred Broy. Compositional Refinement of Interactive Systems Modelled by Relations. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 1997.

[Bro97b] Manfred Broy. Refinement of time. *Transformation-Based Reactive Systems Development*, Volume 1231/1997:44–63, 1997.

[Bro05] Manfred Broy. *Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures–The Janus-Approach*, pages 47–81. Springer, 2005.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[BS10] Hamid Bagheri and Kevin J. Sullivan. Monarch: Model-based development of software architectures. In Petriu et al. [PRH10], pages 376–390.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[BSS10] Hamid Bagheri, Yuanyuan Song, and Kevin J. Sullivan. Architectural style as an independent variable. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 159–162. ACM, 2010.

[BWH10] Nelis Boucké, Danny Weyns, and Tom Holvoet. Composition of architectural models: Empirical analysis and language support. *Journal of Systems and Software*, 83(11):2108–2127, 2010.

[BYN+11] Razieh Behjati, Tao Yue, Shiva Nejati, Lionel C. Briand, and Bran Selic. Extending SysML with AADL concepts for comprehensive system architecture modeling. In Robert B. France, Jochen Malte Küster, Behzad Bordbar, and Richard F. Paige, editors, *ECMFA*, volume 6698 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2011.

[CBB+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2 edition, 2010.

[CCG+03] Ping Chen, Matt Critchlow, Akash Garg, Christopher van der Westhuizen, and André van der Hoek. Differencing and merging within an evolving product line architecture. In *PFE*, volume 3014 of *LNCS*. Springer, 2003.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer, 2009.

[CL07] Christos G. Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer, 2007.

[Cle96] Paul Clements. A Survey of Architecture Description Languages. In *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*, pages 16–25, 1996.

[CRT07] Paul Caspi, Pascal Raymond, and Stavros Tripakis. Synchronous programming. In I. Lee, J. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*, chapter 14, pages 1–21. Chapman & Hall, 2007.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *FMSD*, 19(1):45–80, 2001.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DvdHT01] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *WICSA* [IEE01], pages 103–112.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Computer-Aided Verification, (CAV '98)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, 1998.

[FBDCS11] Michalis Famelis, Shoham Ben-David, Marsha Chechik, and Rick Salay. Partial models: a position paper. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 1:1–1:4, New York, NY, USA, 2011. ACM.

[FG12] Peter H. Feiler and David P. Gluch. *Model-based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. SEI Series in Software Engineering. Addison-Wesley Longman, Amsterdam, 2012.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, Software Engineering Institute, Carnegie Mellon University, 2006.

[FMS11] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML*. Morgan Kaufmann, 2011.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based modeling of function nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop, Paderborn,*, October 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-centric modeling of automotive logical architectures. In *Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of ERTS '08*, 2008.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, pages 76–89, March 2008.

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability Solvers. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[GR06] Boris Gajanovic and Bernhard Rumpe. Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Informatik-Bericht 2006-03, Technische Universität Braunschweig, Carl-Friedrich-Gauss-Fakultät für Mathematik und Informatik, 2006.

[GR07] Borislav Gajanovic and Bernhard Rumpe. Alice: An advanced logic for interactive component engineering. In *4th International Verification Workshop (Verify'07)*, Bremen, 2007.

[GR10] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In Radu Calinescu and Ethan K. Jackson, editors, *Monterey Workshop*, volume 6662 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2010.

[Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten.* PhD thesis, RWTH Aachen University, 2010.

[Gru05] Lars Grunske. Formalizing architectural refactorings as graph transformation systems. In Lawrence Chung and Yeong-Tae Song, editors, *SNPD*, pages 324–329. IEEE Computer Society, 2005.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Vincenzo Ambriola and Genoveffa Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, chapter 1, pages 1–39. World Scientific Pub Co Inc, 1993.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, 1994.

[GSB98] Radu Grosu, Thomas Stauner, and Manfred Broy. A Modular Visual Model for Hybrid Systems. In Anders Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume

1486 of *Lecture Notes in Computer Science*, pages 471–471. Springer Berlin / Heidelberg, 1998.

[GV06] Holger Giese and Alexander Vilbig. Separation of non-orthogonal concerns in software architecture and design. *Software and Systems Modeling*, 5(2):136–169, 2006.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[HB06] Louis Hugues and Nicolas Bredeche. Simbad: An Autonomous Robot Simulation Package for Education and Research. In Stefano Nolfi, Gianluca Baldassarre, Raffaele Calabretta, John C. T. Hallam, Davide Marocco, Jean-Arcady Meyer, Orazio Miglino, and Domenico Parisi, editors, *SAB*, volume 4095 of *Lecture Notes in Computer Science*, pages 831–842. Springer, 2006.

[HF07] Florian Hölzl and Martin Feilkas. Autofocus 3 - a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *LNCS*, pages 317–322. Springer, 2007.

[HK02] David Harel and Orna Kupferman. On Object Systems and Behavioral Inheritance. *IEEE Trans. Software Eng.*, 28(9):889–903, 2002.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 61 –66, june 2012.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *SoSyM*, 7(2):237–252, 2008.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004.

[HR07] Grégoire Hamon and John M. Rushby. An operational semantics for stateflow. *STTT*, 9(5-6):447–456, 2007.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, pages 13–22, Munich, Germany, February 2010. fortiss GmbH.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John, and Klaus Schmid, editors, *SPLC*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernard Rumpe. Montiarc - architectural modeling of interactive distributed and cyber-physical systems. Technical Report AIB-2012-03, RWTH Aachen, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors, *MBEES*, pages 1–10. fortiss GmbH, München, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving delta-oriented software product line architectures. In Radu Calinescu and David Garlan, editors, *Monterey Workshop*, volume 7539 of *Lecture Notes in Computer Science*, pages 183–208. Springer, 2012.

[HS97] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch*, pages 343 – 352. GMD Verlag (St. Augustin), 1997.

[HS00] Thomas Hune and Anders Sandholm. A case study on using automata in control synthesis. In T. S. E. Maibaum, editor, *FASE*, volume 1783 of *Lecture Notes in Computer Science*, pages 349–362. Springer, 2000.

[HS12] David Harel and Itai Segall. Synthesis from scenario-based specifications. *J. Comput. Syst. Sci.*, 78(3):970–980, 2012.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent graphical specification of distributed systems. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 122–141, 1997.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus — a tool for distributed systems specification. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 467–470. Springer Berlin / Heidelberg, 1996.

[HST10] Florian Hoelzl, Maria Spichkova, and David Trachtenherz. AutoFocus Tool Chain. Technical Report TUM-I1021, TU Munich, 2010.

[Huf12] Brian Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012.

[HWF+10] Jorgen Hansson, Lutz Wrage, Peter H. Feiler, John Morley, Bruce A. Lewis, and Jérôme Hugues. Architectural Modeling to Verify Security and Nonfunctional Behavior. *IEEE Security & Privacy*, 8(1):43–49, 2010.

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo O. Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University), 2004.

[IEE] IEEE. IEEE 1471-2000: Recommended Practice for Architectural Description for Software-Intensive Systems. `http://standards.ieee.org/findstds/standard/1471-2000.html`.

[IEE01] IEEE Computer Society. *2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*. IEEE Computer Society, 2001.

[IKL+00] Torsten K. Iversen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-checking real-time control programs: verifying lego(r) mindstorms^{tm} systems using uppaal. In *ECRTS*, pages 147–155. IEEE Computer Society, 2000.

[IKZ02] Valérie Issarny, Christos Kloukinas, and Apostolos Zarras. Systematic aid for developing middleware architectures. *Commun. ACM*, 45(6):53–58, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[JLB11] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 653–667. Springer, 2011.

[JPL+11] Henk Jonkers, Erik Proper, Marc M. Lankhorst, Dick A. C. Quartel, and Maria-Eugenia Iacob. Archimate(r) for integrated modelling throughout the architecture development and implementation cycle. In Birgit Hofreiter, Eric Dubois, Kwei-Jay Lin, Thomas Setzer, Claude Godart, Erik Proper, and Lianne Bodenstaff, editors, *CEC*, pages 294–301. IEEE, 2011.

[JS00] Daniel Jackson and Kevin J. Sullivan. COM revisited: tool-assisted modelling of an architectural framework. In *SIGSOFT FSE*, pages 149–158. ACM, 2000.

[KA03] Frank Klassner and Scott D. Anderson. LEGO MindStorms: not just for K-12 anymore. *IEEE Robot. Automat. Mag.*, 10(2):12–18, 2003.

[KC94] Paul Kogut and Paul Clements. Features of architecture description languages. In *In Proceedings of the Eighth International Workshop on Software Specification and Design*, pages 16–25, 1994.

[KG10] Jung Soo Kim and David Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83(7):1216–1235, 2010.

[KI01] Christos Kloukinas and Valérie Issarny. SPIN-ning Software Architectures: A Method for Exploring Complex. In *WICSA* [IEE01], pages 67–76.

[Kir11] Dennis Kirch. Analysis of Behavioral Specifications for Distributed Interactive Systems with MONA. Bachelor Thesis, RWTH Aachen University, 2011.

[KJ09] Seung Han Kim and Jae Wook Jeon. Introduction for Freshmen to Embedded Systems Using LEGO Mindstorms. *IEEE Trans. Education*, 52(1):99–108, 2009.

[KNS96a] Nils Klarlund, Mogens Nielsen, and Kim Sunesen. Automated logical verification based on trace abstractions. In James E. Burns and Yoram Moses, editors, *PODC*, pages 101–110. ACM, 1996.

[KNS96b] Nils Klarlund, Mogens Nielsen, and Kim Sunesen. A case study in automated verification based on trace abstractions. In Manfred Broy, Stefan Merz, and Katharina Spies, editors, *Formal System Specification, The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*, pages 341–374. Springer Verlag, 1996.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, 2010.

[KRPP09] Dimitrios S. Kolovos, Davide Di Ruscio, Richard F. Paige, and Alfonso Pierantonio. Different Models for Model Matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA, May 2009. IEEE Computer Society.

[Kru95] Philippe Kruchten. Architectural Blueprints – The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, 1995.

[Krü11] Andreas Krüger. Stream-Based Specification in Isabelle/HOLCF – Towards ALICE 2.0. Bachelor Thesis, RWTH Aachen University, 2011.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *STTT*, 12(5):353–372, 2010.

[Lar89] Kim Guldstrand Larsen. Modal specifications. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *ISORC*, pages 363–369. IEEE Computer Society, 2008.

[Lee09] Edward A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009.

[Lee10] Edward A. Lee. Disciplined heterogeneous modeling - invited paper. In Petriu et al. [PRH10], pages 273–287.

[LML06] Xiaojun Liu, Eleftherios Matsikoudis, and Edward A. Lee. Modeling timed concurrent systems. In *CONCUR*, pages 1–15, 2006.

[LNW07a] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal i/o automata for interface and product line theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.

[LNW07b] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. On modal refinement and consistency. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2007.

[LPJ10] Marc M. Lankhorst, Henderik Alex Proper, and Henk Jonkers. The anatomy of the archimate language. *IJISMD*, 1(1):1–32, 2010.

[LS11] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems — A Cyber-Physical Systems Approach*. LeeSeshia.org, first edition, version 1.08 edition, 2011.

[LSV03] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Hybrid i/o automata. *Inf. Comput.*, 185(1):105–157, 2003.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[LT89] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, September 1989.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society, 1990.

[Mar12] Philip Martzok. Action-Stream-Based Modeling of Robots using I/O$^{\omega}$-automata and MontiArc. Bachelor Thesis, RWTH Aachen University, 2012.

[MC12] Alvaro Miyazawa and Ana Cavalcanti. Refinement-oriented models of stateflow charts. *Science of Computer Programming*, 77(10–11):1151 – 1177, 2012.

[McM99] Kenneth L. McMillan. The SMV language. Technical report, Cadence Berkeley Labs, 1999.

[MDT07] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information & Software Technology*, 49(1):12–31, 2007.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34:1045–1079, 1955.

[Mey75] Albert R. Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer Berlin Heidelberg, 1975.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Software Eng.*, 39(6):869–891, 2013.

[MMP91] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *REX Workshop*, pages 447–484, 1991.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving software architecture descriptions of critical systems. *IEEE Computer*, 43(5):42–48, 2010.

[Mon98] Robert T. Monroe. Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, 1998.

[Mon99] Robert T. Monroe. *Rapid Development of Custom Software Architecture Design Environment*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of concurrent and reactive systems: specification*. Springer, 1992.

[MQR95] Mark Moriconi, Xiaolei Qian, and Robert A. Riemenschneider. Correct architecture refinement. *IEEE Trans. Software Eng.*, 21(4):356–372, 1995.

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal object diagrams. In Mira Mezini, editor, *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 281–305. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of component and connector models from crosscutting structural views. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/SIGSOFT FSE*, pages 444–454. ACM, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *ICSE*, pages 95–105. ACM, 2014.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE*, pages 44–53, 1999.

[MS96] Olaf Müller and Peter Scholz. Specification of Real-Time and Hybrid Systems in FOCUS. Technical Report TUM-I9627, Technische Univerität München, 1996.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[NBD09] Andrew L. Nelson, Gregory J. Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[Obj12a] Object Management Group. OMG Unified Modeling Language (OMG UML). `http://www.uml.org/`, 2012. Accessed 8/2012.

[Obj12b] Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML), Version 1.3. `http://www.omg.org/spec/SysML/1.3/`, 2012. Accessed 8/2012.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society, 1990.

[PR97] Barbara Paech and Bernhard Rumpe. State based service description. In *FMOODS '97: Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 293–302, London, UK, UK, 1997. Chapman & Hall, Ltd.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe And Filter Architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.

[PRH10] Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors. *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *LNCS*. Springer, 2010.

[Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. Accessed 10/13, cited like this on request by the authors.

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[Rab63] Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[Rac13] Deni Raco. Towards ALICE 2.0, a Logic for Stream Processing Functions. Diploma Thesis, RWTH Aachen University, 2013.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 5(1-2):29–53, July 2011.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In *Modelling and Quality in Requirements Engineering*. Monsenstein und Vannerdat Münster, 2012.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany*, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In Stefan Wagner and Horst Lichter, editor, *Software Engineering 2013 Workshopband*, LNI, pages 155–170. GI, Köllen Druck+Verlag GmbH, Bonn, 2013.

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 6-10 2013.

[RRW14] Jan Oliver Ringert, Bernard Rumpe, and Andreas Wortmann. Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Technical Report AIB-2014-XXX, RWTH Aachen, 2014. Draft available from [wwwt].

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, 2 edition, 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML*. Springer, 2 edition, 2012.

[RW10] Lukman Ab Rahim and Jon Whittle. Verifying semantic conformance of state machine-to-java code generators. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2010.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[Sch98] Peter Scholz. A refinement calculus for statecharts. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 285–301. Springer Berlin Heidelberg, 1998.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[Sch09] Bernhard Schätz. *Model-Based Development of Software Systems: From Models to Tools*. Technische Universität München, 2009. Habilitation Thesis.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SE04] Mehrdad Sabetzadeh and Steve M. Easterbrook. An Algebraic Framework for Merging Incomplete and Inconsistent Views. Technical Report CSRG-496, Department of Computer Science, University of Toronto, September 2004.

[SE06] Mehrdad Sabetzadeh and Steve Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3):174–193, 2006.

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2012.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[SG04] Bradley R. Schmerl and David Garlan. Acmestudio: Supporting style-centered architecture development. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *ICSE*, pages 704–705. IEEE Computer Society, 2004.

[Slo97] Oscar Slotosch. *Refinements in HOLCF: Implementation of interactive systems*. PhD thesis, Technische Universität München, 1997.

[SP10a] Bernhard Schätz and Christian Pfaller. Test case integration: From components to systems. In Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors, *MBEES*, pages 65–76. fortiss GmbH, München, 2010.

[SP10b] Bernhard Schätz and Christian Pfaller. Integrating component tests to system tests. *Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008). Electronic Notes in Theoretical Computer Science*, 260(0):225 – 241, 2010.

[Spi92] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[Spi07] Maria Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, Technische Universität München, 2007.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, New York, 1973.

[Ste97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[SVB+06] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. Pitman, 2006.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[The12] The Open Group. *ArchiMate® 2.0 Specification*. Van Haren Publishing, 2012.

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. Elsevier Science & Technology, 1990.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[Tra09] David Trachtenherz. *Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme*. PhD thesis, Institut für Informatik, Technische Universität München, 2009.

[TSSL13] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A. Lee. A modular formal semantics for ptolemy. *Mathematical Structures in Computer Science*, 23:834–881, 8 2013.

[UBC07] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, pages 34–43, 2007.

[Vö11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1998.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[Wei07] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, 2007.

[WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323. ACM, 2000.

[wwwa] AADL website. http://www.aadl.info/. Accessed 8/2013.

[wwwb] Alloy Analyzer website. http://alloy.mit.edu/. Accessed 8/2012.

[wwwc] AutoFOCUS 3 – The Picture Book. http://www4.in.tum.de/~ccts/af3/picturebook/af3_picturebook.pdf. Accessed 01/13.

[wwwd] AutoFocus3 developers website. https://af3.fortiss.org/projects/autofocus3. Accessed 8/2012.

[wwwe] AutoFocus3 website. http://autofocus.informatik.tu-muenchen.de/. Accessed 8/2012.

[wwwf] FreeMarker website. http://freemarker.sourceforge.net/. Accessed 7/2012.

[wwwg] IBM Rational Rhapsody family. http://www-03.ibm.com/software/products/us/en/ratirhapfami. Accessed 11/13.

[wwwh] Java for LEGO Mindstorms. http://www.lejos.org. Accessed 10/13.

[wwwi] JUnit – A programmer-oriented testing framework for Java. http://junit.org. Accessed 10/13.

[wwwj] LEGO Mindstorms Education NXT User Guide. http://cache.lego.com/r/education/-/media/lego%20education/home/downloads/user%20guides/global/mindstorms/9797_lme_userguide_us_low.pdf?l.r=-1708348262. Accessed 10/13.

[wwwk] LEGO MINDSTORMS NXT Hardware – MATLAB & Simulink Website. http://www.mathworks.com/help/simulink/lego-mindstorms-nxt.html. Accessed 11/13.

[wwwl] LEGO Mindstorms NXT website. http://mindstorms.lego.com/. Accessed 8/2012.

[wwwm] LEGO.com Mindstorms. http://mindstorms.lego.com. Accessed 10/13.

[wwwn] MathWorks Simulink website. http://www.mathworks.com/products/simulink/. Accessed 8/2012.

[wwwo] Mathworks stateflow. http://www.mathworks.com/products/stateflow/index.html. Accessed 10/2013.

[wwwp] MiniSat website. http://minisat.se/. Accessed 7/2012.

[wwwq] MontiArc website. http://www.monticore.de/languages/montiarc/. Accessed 8/2012.

[wwwr] MontiArcAutomaton Code Generation website. http://www.monticore.de/languages/montiarcautomaton/codegen/. Contains supporting materials for this thesis.

[wwws] MontiArcAutomaton Verification website. http://www.monticore.de/languages/montiarcautomaton/verification/. Contains supporting materials for this thesis.

[wwwt] MontiArcAutomaton website. http://www.monticore.de/languages/montiarcautomaton/. Contains supporting materials for this thesis.

[wwwu] MontiArcView Verification and Synthesis Evaluation Materials. http://www.se-rwth.de/materials/cncviews/. Contains supporting materials for this thesis.

[wwwv] MontiCore Project website. http://www.monticore.org/. Accessed 8/2012.

[wwww] NuSMV home page. http://nusmv.fbk.eu/. Accessed 01/13.

[wwwx] NXT Programs Bumper Car website. http://www.nxtprograms.com/bumper_car/. Accessed 8/2012.

[wwwy] The ACME Studio Homepage. http://www.cs.cmu.edu/~acme/AcmeStudio/. Accessed 8/2012.

[wwwz] The MONA Project. http://www.brics.dk/mona/. Accessed 09/2013.

[wwwaa] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/. Accessed 10/13.

[wwwab] Visual Paradigm for UML Website. http://www.visual-paradigm.com/. Accessed 11/2013.

[XS07] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.

Index

List of Definitions

List of Figures

List of Listings

Appendix A.

Symbols

General symbols

$\triangle$	end of a definition
$\square$	end of a proof

Automata

$\varepsilon_{\diamond}$	absence of a message in MAA_{ts} automata
$*$	omission of elements in the syntax of ${}^{*}\mathrm{MAA}_{ts}$ automata
$\vec{I}$	cross product of sets, e.g., $\vec{I} = I_1 \times \ldots \times I_n$
$\vec{i}$	tuple; element of cross product of sets, e.g., $\vec{i} = (i_1, \ldots, i_n) \in \vec{I} = I_1 \times \ldots \times I_n$

Logic

$\forall$	universal quantification
$\exists$	existential quantification
$\exists_!$	existential quantification satisfied by exactly one element
THE	definite description operator; $THE\ x : \Phi(x)$ is the unique x that satisfies Φ

Arithmetic for $a, b \in \mathbb{N}$

$a * b$	multiplication
$\Pi_{r \in R} f(r)$	multiplication of values $f(r)$ for every $r \in R$

Relations

R^{+}	transitive closure of the binary relation R
R^{-1}	transposed relation, e.g., inverse of a binary relation R
$R \bowtie Q$	join of relations on the last element of R and first element of Q removing the matching element: $(x_1, .., x_{m-1}, x_{m+2}, .., x_n) \in R \bowtie Q \Leftrightarrow (x_1, .., x_m) \in R \wedge (x_{m+1}, .., x_n) \in Q \wedge x_m = x_{m+1}$ equivalent to the dot-join operator of Alloy [Jac06, Section 3.4.3]

Sets

$\mathbb{N}$	natural numbers including 0
$\mathbb{N}_\infty$	$\mathbb{N} \cup \{\infty\}$
$\mathbb{R}$	real numbers
$S \subseteq T$	the set S is included in the set T
$S \times T$	the cross product of sets S and T
$\times_{r \in S}(f(r))$	the cross product of all sets $f(r)$ (using a fixed order of the elements $r \in S$)
$S \cup T$	the union of sets S and T
$\bigcup_{r \in S}(f(r))$	the union of all sets $f(r)$
$S \cap T$	the intersection of sets S and T
$S \setminus T$	the removal of all elements of T from S

Streams where $n \in \mathbb{N}$, M set of messages, $s \in M^\omega$

M^ω	finite or infinite stream
M^∞	infinite (timed) stream
$\vec{I}$	input type for component type *cmp* with $\vec{I} = \times_{p \in cmp.CPorts_{IN}}(p.type)$
$\vec{O}$	output type for component type *cmp* with $\vec{O} = \times_{p \in cmp.CPorts_{OUT}}(p.type)$
$\langle\rangle$	empty stream
$m{:}s$	append first element
$s \frown s'$	concatenation of streams
$s \sqsubseteq s'$	prefix relation
$\#s \in \mathbb{N}_\infty$	length of stream
$s.n$	n^{th} element of stream
$s\|_n$	prefix of length n
m^n	message iterated n times
s^n	stream iterated n times
$s\|_P$	domain restriction of input or output s of a component $cmp \in CTDefs$ to ports in $P \subseteq cmp.CPorts$: $s\|_P = \times_{p \in P}(s_p)$

C&C Views

$\perp$	unknown ports, names, or types in C&C views
$\cong$	correspondence between ports from a C&C model and a C&C view (see Definition 3.8, Item 4 (b))

Appendix B.

Translation Rule Notation

Many analyses on models can be carried out by translating models from one language into another language that provides similar analyses and then translating the results back into the problem domain. The modeling languages employed in this thesis have a textual concrete syntax. We use a compact notation for translation rules from the abstract syntax of one language into the concrete syntax of a target language.

The translations described in this thesis are implemented using FreeMarker templates [wwwf] following the MontiCore code generation approach described in [Sch12]. In this approach, templates retrieve relevant information about the source model from nodes of the model's abstract syntax tree and calculators implemented in Java.

We use a more compact notation introduced here, that allows us to (1) use less technical and conceptually simplified source structures, (2) access information about the relevant input structures independent of current AST nodes, and (3) declaratively express complex calculation that otherwise require Java implementations of calculators in the MontiCore code generation framework.

In the following sections we introduce the basic notations and constructs used in the translation rules. The translation constructs are illustrated on example applications to a model of the component `BumpControl` shown in Listing 6.5. We show excerpts of FreeMarker templates to illustrate our translation notation for users of the MontiCore code generation framework. For a detailed reference of this framework see [Sch12].

Source Structure and Target Syntax

In the translation rules, we identify and access elements of the source structure using *italics*. Elements of the concrete syntax of the target language are typeset in `underlined type writer font`.

As an example, consider a translation of the MontiArcAutomaton component type definitions defined in Definition 6.8 into Java. In the abstract syntax, a component type definition *cmp* has the component type name *cmp.cType*. In a translation into the target language Java, the translation rule shown in Figure B.1 creates a Java class declaration with the name of the component.

Translation Rule

Translation rule:

R1 `class` *cmp.cType* `implements Component {`

```
  /*...*/
}
```

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1 class BumpControl implements Component {
2   /* ... */
3 }
```

Figure B.1.: Rule to create a class in concrete Java syntax for a component *cmp* with name *cmp.cType*.

In FreeMarker templates the current node of the abstract syntax tree of the source model is available as the variable `ast`. The freemarker syntax to access variables and invoke methods is `${...}`. The example in Figure B.2 invokes the method `printName()` of the AST class `ASTArcComponent` which returns a string. Often the templates refer to manually implemented methods of the AST nodes that allow convenient printing of complex AST nodes.

FreeMarker Template

```
class ${ast.printName()} implements Component {
  /* ... */
}
```

Figure B.2.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.1.

Quantification

For multiple repetitions of fragments we use quantification over the elements of the source structure that the fragment is repeated for. Quantification over elements introduces the quantified variables in the enclosed lines and produces one copy for each element. An example is shown in Figure B.3 where a variable p is introduced for ports from the set of ports $cmp.CPorts$.

Translation Rule

Translation rule:

R2 $\forall p \in cmp.CPorts$:
`private Port<`$p.type$`> my_`$p.name$`;`

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1 private Port<Boolean> my_bump;
2 private Port<TimerSignal> my_ts;
3 private Port<TimerCmd> my_tc;
4 private Port<MotorCmd> my_rMot;
5 private Port<MotorCmd> my_lMot;
```

Figure B.3.: Rule to create a variable declaration for every port p in $cmp.CPorts$.

The quantification used here is similar to the `foreach` directive of the FreeMarker template language as shown in Figure B.4.

FreeMarker Template

```
<#foreach port in ast.getPorts()>
  private Port<${port.printType()}> my_${port.printName()};
</#foreach>
```

Figure B.4.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.3.

Iteration

The iteration of the statement *body* for elements *elems* with an infix separator *sep* is specified inline using the operator $\{\!|body|\!\}^{sep}_{elems}$. An example application of the operator is shown in Figure B.5. The inner translation rule $\llcorner$`"`$\lrcorner\, p.name \,\llcorner$`"`$\lrcorner$ prints the name of every port $p \in cmp.CPorts$ in quotation marks and joins the separate elements with the Java String concatenation operator $\overset{+}{\llcorner\lrcorner}$.

Translation Rule

Translation rule:

R3 `public String getPortNames() {`

`return` $\{\!|$ `"`$\lrcorner\, p.name \,\llcorner$`"`$|\!\}^{+}_{p \in cmp.ports}$ `;`

`}`

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1 public String getPortNames() {
2   return "bump" + "ts" + "tc" + "rMot" + "lMot";
3 }
```

Figure B.5.: Rule to create a Java method that returns all port names of a component. Strings of port names are concatenated in Java using $\overset{+}{\llcorner\lrcorner}$.

The iteration $\{\!|body|\!\}^{sep}_{elems}$ can be implemented as a combination of the `list` and `_has_next` directives of FreeMarker as shown in Figure B.6.

FreeMarker Template

```
public String getPortNames() {
  return <#list port in ast.getPorts()> "${port.printName()}"
                  <#if port_has_next> + </#if></#list>;
}
```

Figure B.6.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.5.

let–in

The construct **let** *var* = *exp* **in** *body* assigns the evaluation of the expression *exp* to the variable *var* and executes the *body* of the rule that might depend on the variable *var*. An example of the application of the construct **let** *var* = *exp* **in** *body* is shown in Figure B.7.

Translation Rule

Translation rule:

R4 **let** $inPorts = \{p \in cmp.CPorts \mid p.dir = IN\}$ **in**
$\forall p \in inPorts$:
`public void setPort_`*p.name*`(Port<`*p.type*`> port) {`
`  my_`*p.name* `= port;`
`}`

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1 public void setPort_bump(Port<Boolean> port) {
2   my_bump = port;
3 }
4
5 public void setPort_ts(Port<TimerSignal> port) {
6   my_ts = port;
7 }
```

Figure B.7.: Rule to create Java methods to set the instances of incoming ports.

The MontiCore code generation framework provides a calculator mechanism to do complex calculations in Java. Calculators allow the definition of variables accessible in FreeMarker templates. A corresponding example with a Java calculator `Ports-Calculator` is shown in Figure B.8.

FreeMarker Template

```
<#if op.callCalculator(
        "montiarcautomaton.codegen.PortsCalculator")>
  Java calculator stores set of incoming ports in variable inPorts
  (successful calculators return true)
  <#foreach port in op.getValue(inPorts)>
  public void setPort_${port.printName()}(
                      Port<${port.printType()}> port) {
    my_${port.printName()} = port;
  }
  </#foreach>
</#if>
```

Figure B.8.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.7.

if–then–else

The construct **`if (`***bexp***`) then`** *x* **`else`** *y* evaluates the Boolean expression *bexp* and executes the statement *x* if *bexp* evaluates to **`true`**. The statement *y* is executed if *bexp* evaluates to **`false`**. An example for the application of the **`if-then-else`** construct is shown in Figure B.9.

Translation Rule

Translation rule:

R5
```
public boolean isAtomic() {
if (cmp.CSubCmps ≠ ∅ ) then
  return false;
else
  return true;
}
```

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1 public boolean isAtomic() {
2   return true;
3 }
```

Figure B.9.: Rule R5 generates a method that returns whether the component represented by the generated class is an atomic component.

The template engine FreeMarker provides a similar `if-then-else` construct as shown in Figure B.10.

FreeMarker Template

```
public boolean isAtomic() {
<#if ast.getSubComponents()?has_content>
  return false;
<#else>
  return true;
</#if>
}
```

Figure B.10.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.9.

Executing Rules

The construct **executeRule(**R *param***)** executes the rule R with the parameter *param*. An example for the application of the **executeRule()** construct is shown in Figure B.11. The rule R6 executes the two rules R2 and R4. In this example, no parameters are passed to the rules.

Translation Rule

Translation rule:

R6
```
class cmp.cType extends Component {
  // add variables for all ports
  executeRule(R2)  see rule R2 in Figure B.3
  // add setters for all incoming ports
  executeRule(R4)  see rule R4 in Figure B.7
}
```

Result of application to the component `BumpControl` shown in Listing 6.5:

Java

```
1  class BumpControl implements Component {
2    //add variables for all ports  result of rule R2
3    private Port<Boolean> my_bump;
4    private Port<TimerSignal> my_ts;
5    private Port<TimerCmd> my_tc;
6    private Port<MotorCmd> my_rMot;
7    private Port<MotorCmd> my_lMot;
8
9    //add setters for all incoming ports  result of rule R4
10   public void setPort_bump(Port<Boolean> port) {
11     my_bump = port;
12   }
13   public void setPort_ts(Port<TimerSignal> port) {
14     my_ts = port;
15   }
16
17 }
```

Figure B.11.: Modification of rule R1 that adds the declaration of variables for all ports by executing rule R2 and mutator methods for all input ports by executing rule R4.

In the MontiCore code generation framework the execution of another rule corresponds to the inclusion of a template using the template operator. An example is shown in Figure B.12. For this example we assume that the template shown in Figure B.4 is stored in

the Java package `montiarcautomaton` and has the file name `PortVariables.ftl`. Similarly, the template shown in Figure B.8 is stored in the same package with the file name `PortSetter.ftl`. The template operator includes the templates from these coordinates.

FreeMarker Template

```
class ${ast.printName()} implements Component {
${op.includeTemplate("montiarcautomaton.PortVariables", ast)}
${op.includeTemplate("montiarcautomaton.PortSetter", ast)}
}
```

Figure B.12.: Excerpt of a FreeMarker template implementing the translation rule shown in Figure B.11.

Appendix C.

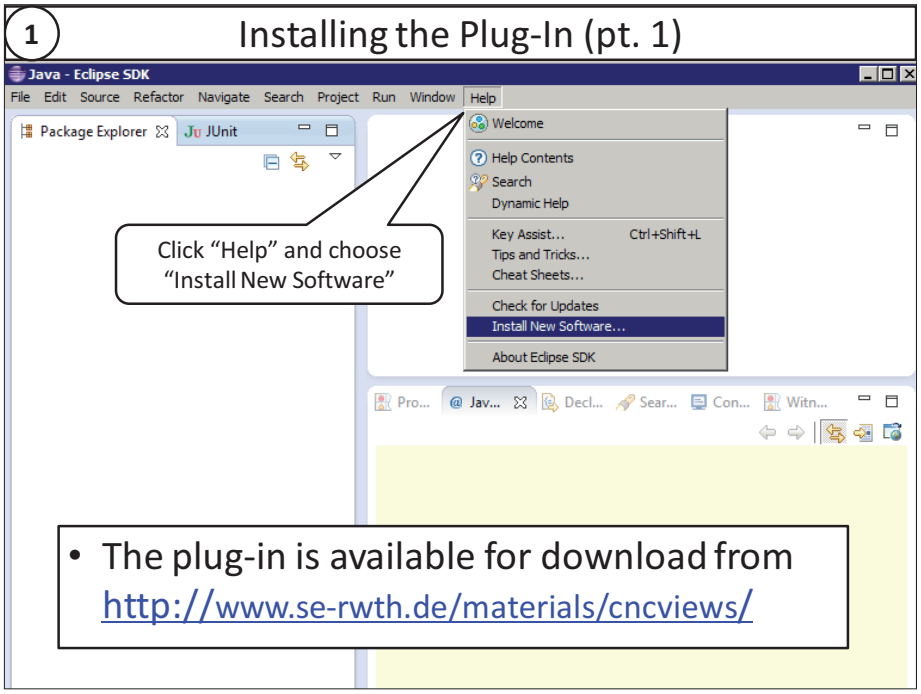
How to Use the C&C Views Verification Plug-In

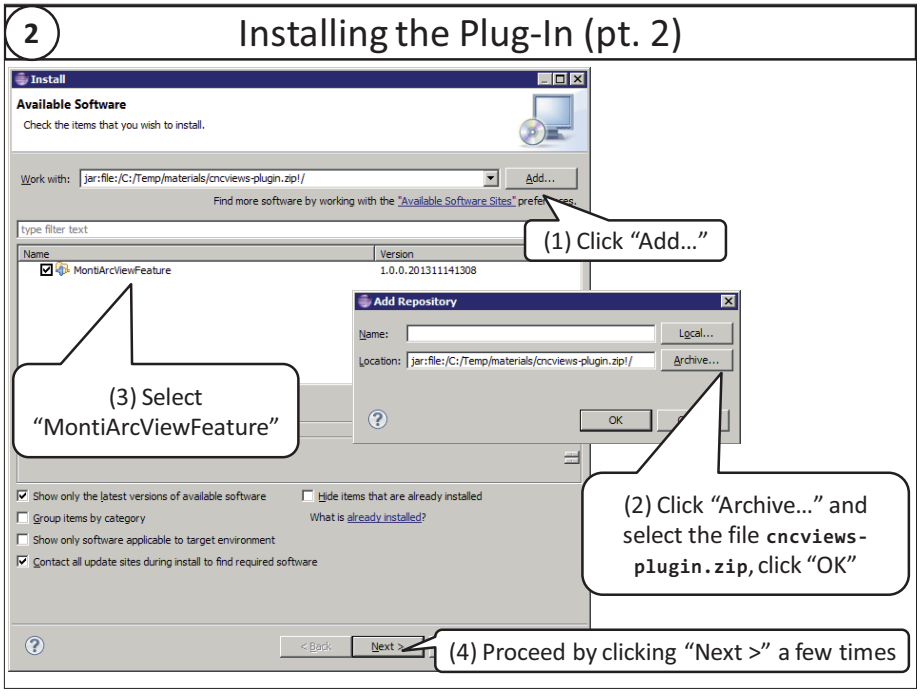
This appendix presents an overview of how to use the C&C views verification Eclipse plug-in. The C&C views verification plug-in is an implementation of the algorithms presented in Chapter 4 for checking C&C views satisfaction and generating witnesses.

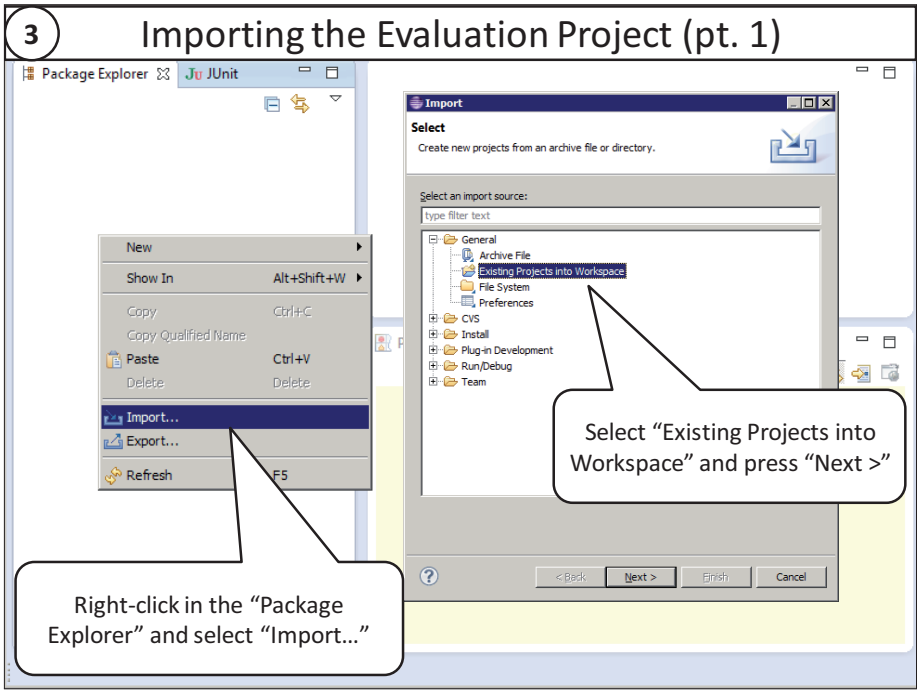
The plug-in comes with the example systems listed in Section 4.4.1 and their corresponding sets of views. In addition we have made available the implementation of the synthetic C&C model generation, view derivation, and the mutations used for our performance analysis described in Section 4.4.2.

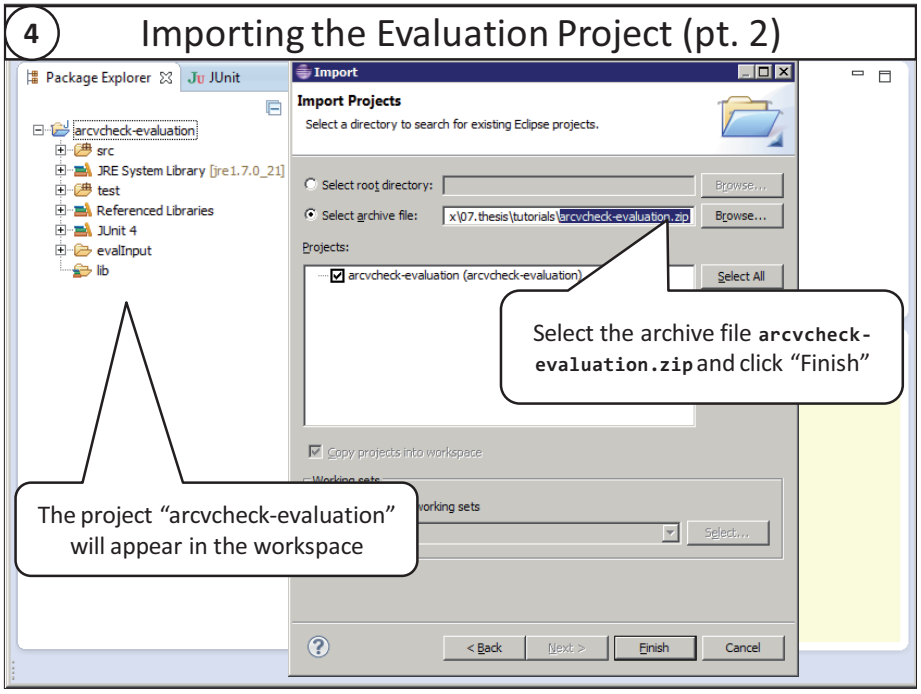
The plug-in and the evaluation project are available for download from [wwwu]. The first slides on the following pages explain how to install the plug-in in Eclipse and how to import the evaluation project. The plug-in has been tested with the following software:

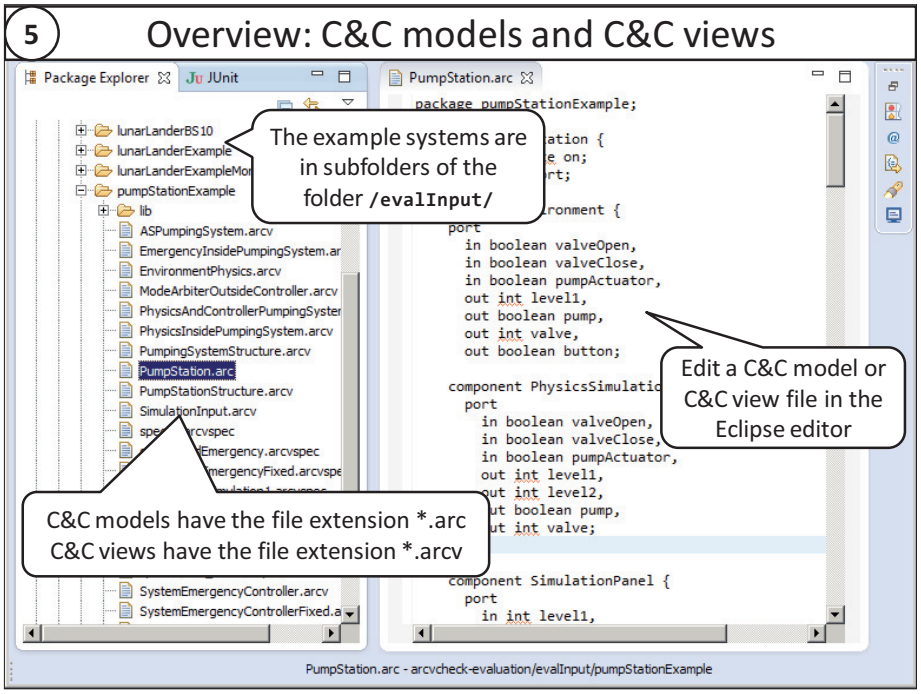
- Eclipse 3.7 (32-bit) and 4.2.2 (32-bit)
- Java SE Development Kit 7, 1.7.0_21 (32-bit)
- Windows 7 (64-bit)

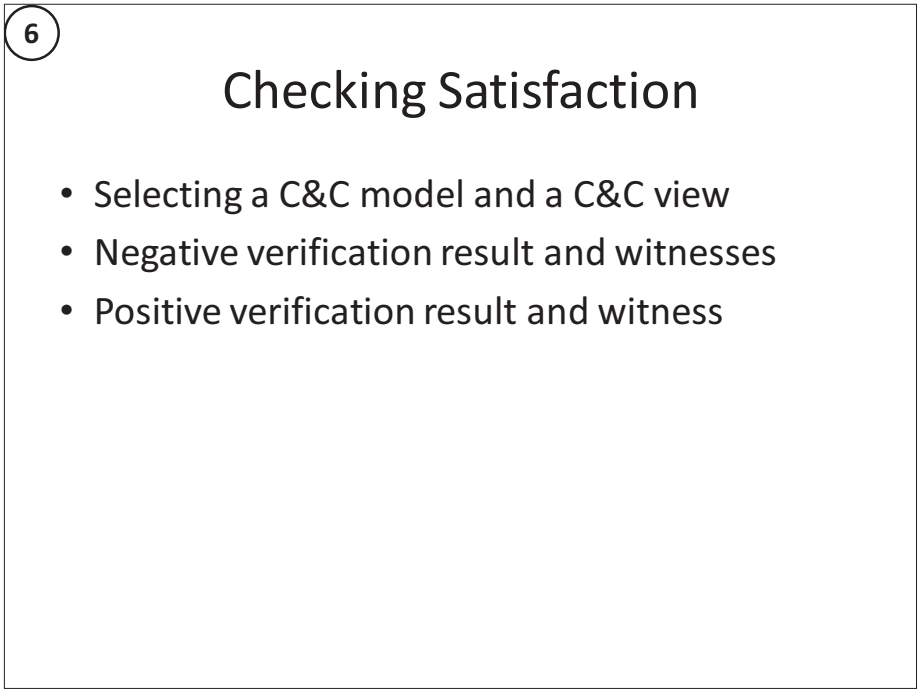

1
Installing the Plug-In (pt. 1)
Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
Welcome
Help Contents
Search
Dynamic Help
Key Assist... Ctrl+Shift+L
Tips and Tricks...
Cheat Sheets...
Check for Updates
Install New Software...
About Eclipse SDK
Click “Help” and choose “Install New Software”
• The plug-in is available for download from http://www.se-rwth.de/materials/cncviews/


2
Installing the Plug-In (pt. 2)
Install
Available Software
Check the items that you wish to install.
Work with: jar:file:/C:/Temp/materials/cncviews-plugin.zip!/
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
MontiArcViewFeature
1.0.0.201311141308
(1) Click “Add...”
Add Repository
Name:
Local...
Location: jar:file:/C:/Temp/materials/cncviews-plugin.zip!/
Archive...
OK
(3) Select “MontiArcViewFeature”
(2) Click “Archive...” and select the file cncviews-plugin.zip, click “OK”
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
(4) Proceed by clicking “Next >” a few times

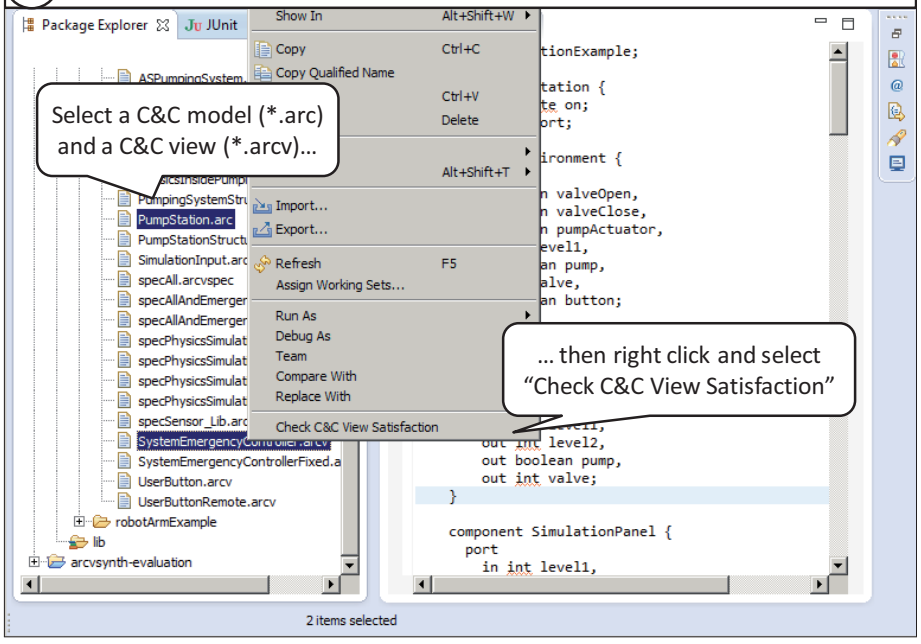

3
Importing the Evaluation Project (pt. 1)
Package Explorer
JUnit
New
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Import...
Export...
Refresh F5
Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
CVS
Install
Plug-in Development
Run/Debug
Team
< Back
Next >
Finish
Cancel
Right-click in the "Package Explorer" and select "Import..."
Select "Existing Projects into Workspace" and press "Next >"


4
Importing the Evaluation Project (pt. 2)
Package Explorer
JUnit
arcvcheck-evaluation
src
JRE System Library [jre1.7.0_21]
test
Referenced Libraries
JUnit 4
evalInput
lib
Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
x\07.thesis\tutorials\arcvcheck-evaluation.zip
Browse...
Projects:
arcvcheck-evaluation (arcvcheck-evaluation)
Select All
Copy projects into workspace
Working sets
Select...
< Back
Next >
Finish
Cancel
The project "arcvcheck-evaluation" will appear in the workspace
Select the archive file arcvcheck-evaluation.zip and click "Finish"

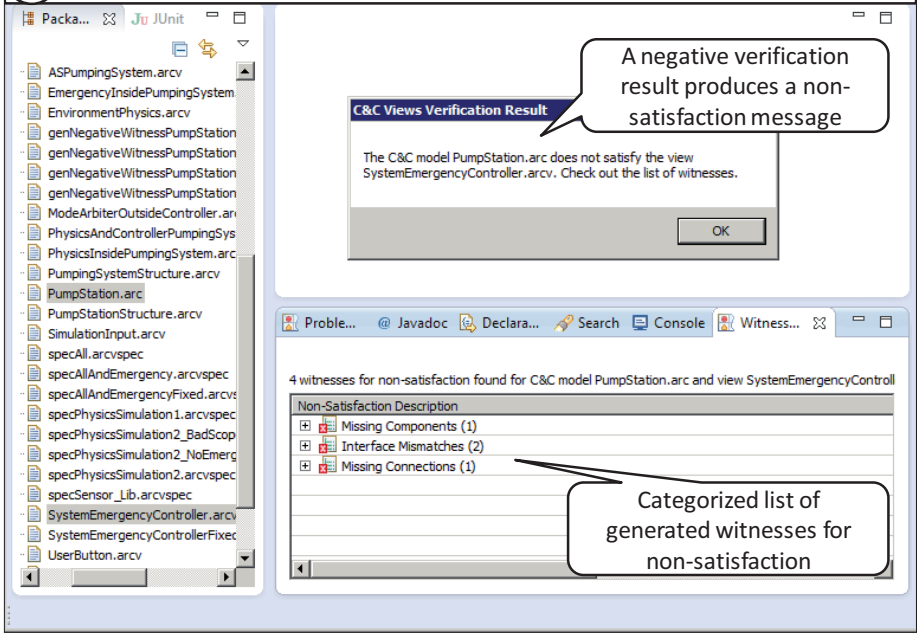

5
Overview: C&C models and C&C views
Package Explorer
JUnit
PumpStation.arc
The example systems are in subfolders of the folder /evalInput/
Edit a C&C model or C&C view file in the Eclipse editor
C&C models have the file extension *.arc
C&C views have the file extension *.arcv
PumpStation.arc - arcvcheck-evaluation/evalInput/pumpStationExample

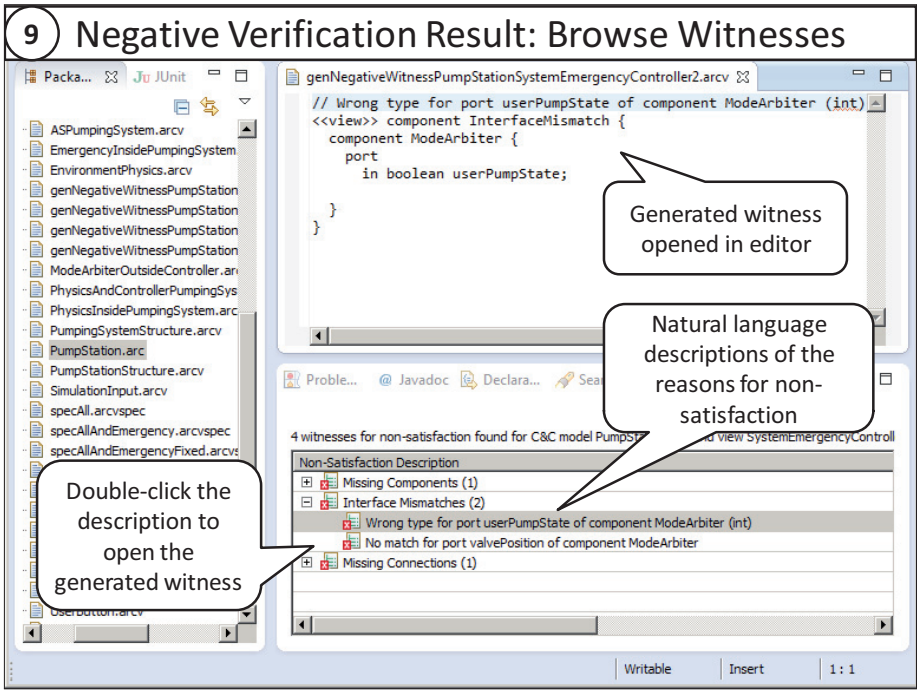

6
Checking Satisfaction
• Selecting a C&C model and a C&C view
• Negative verification result and witnesses
• Positive verification result and witness

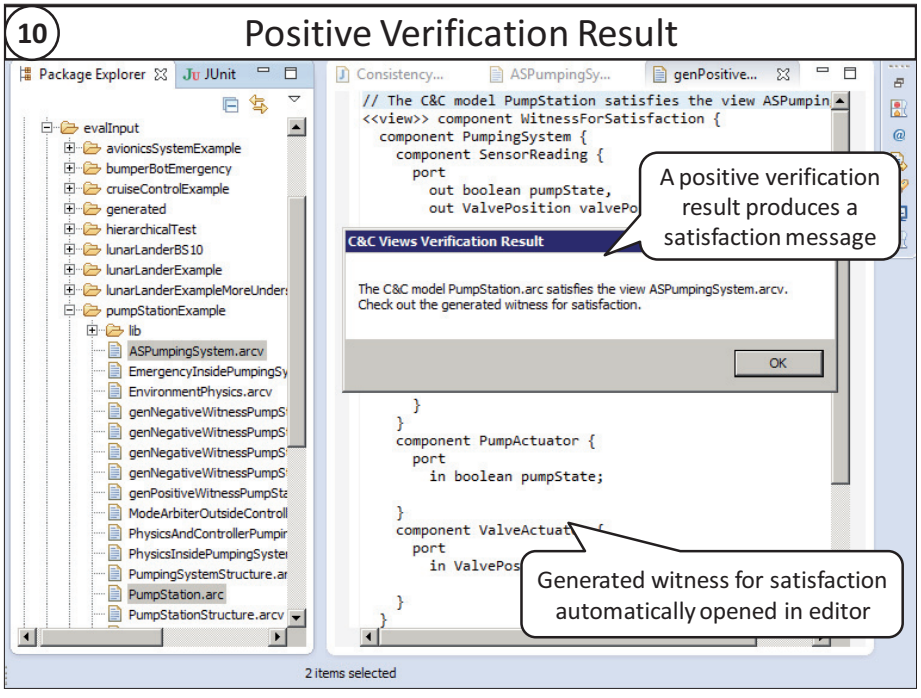
7 Check C&C View Satisfaction



8 Negative Verification Result




9 Negative Verification Result: Browse Witnesses
genNegativeWitnessPumpStationSystemEmergencyController2.arcv
// Wrong type for port userPumpState of component ModeArbiter (int)
<<view>> component InterfaceMismatch {
component ModeArbiter {
port
in boolean userPumpState;
}
}
Generated witness opened in editor
Natural language descriptions of the reasons for non-satisfaction
Double-click the description to open the generated witness
4 witnesses for non-satisfaction found for C&C model PumpSt
Non-Satisfaction Description
Missing Components (1)
Interface Mismatches (2)
Wrong type for port userPumpState of component ModeArbiter (int)
No match for port valvePosition of component ModeArbiter
Missing Connections (1)
Writable
Insert
1 : 1


10 Positive Verification Result
Package Explorer
JUnit
genPositive...
// The C&C model PumpStation satisfies the view ASPumpin
<<view>> component WitnessForSatisfaction {
component PumpingSystem {
component SensorReading {
port
out boolean pumpState,
out ValvePosition valvePo
A positive verification result produces a satisfaction message
C&C Views Verification Result
The C&C model PumpStation.arc satisfies the view ASPumpingSystem.arcv.
Check out the generated witness for satisfaction.
OK
component PumpActuator {
port
in boolean pumpState;
Generated witness for satisfaction automatically opened in editor
2 items selected

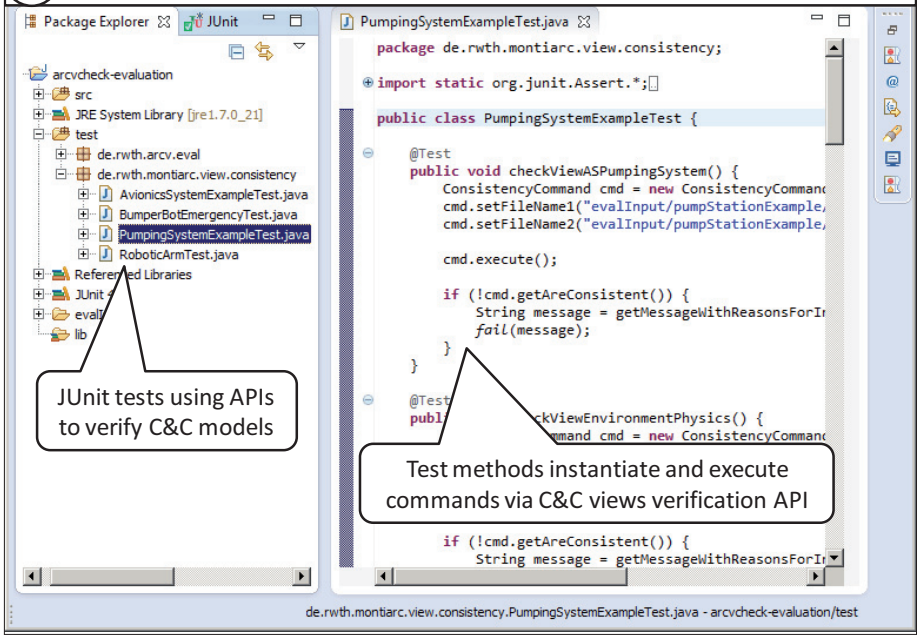
11

Programmatically Executing C&C Views Verification

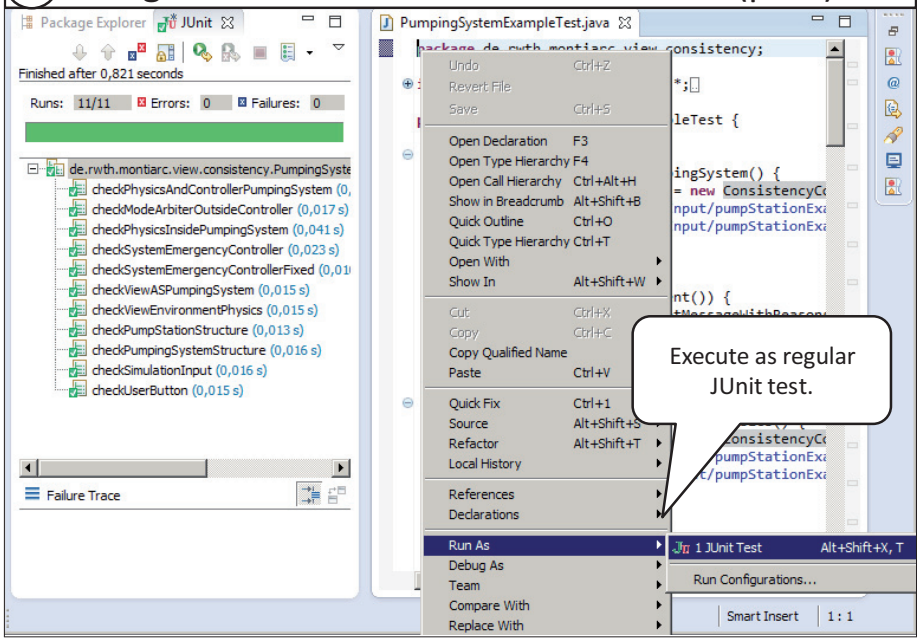
- Executing the verification via JUnit tests, e.g., for regression testing

12

Programmatic Execution of Verification (pt. 1)



13 Programmatic Execution of Verification (pt. 2)

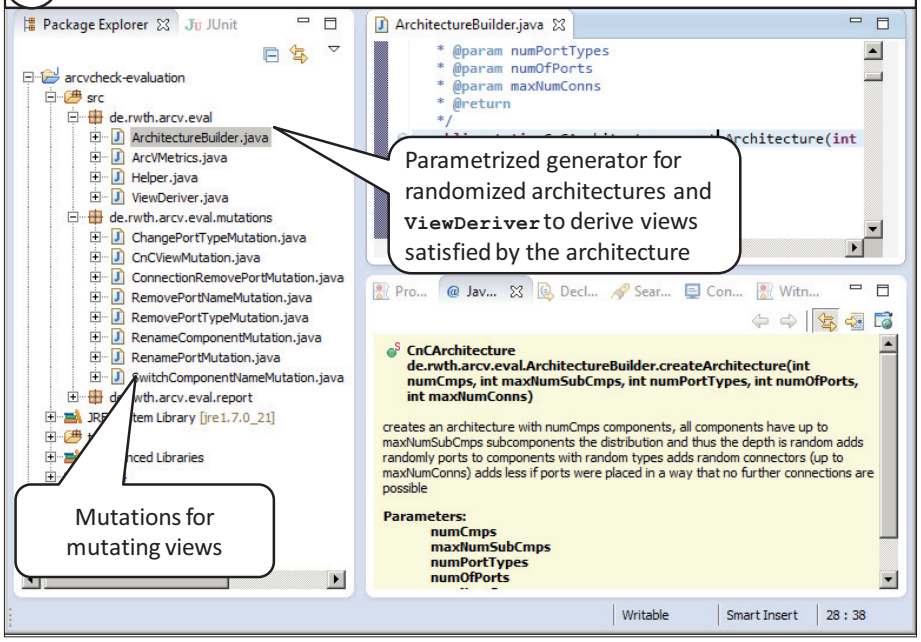


14

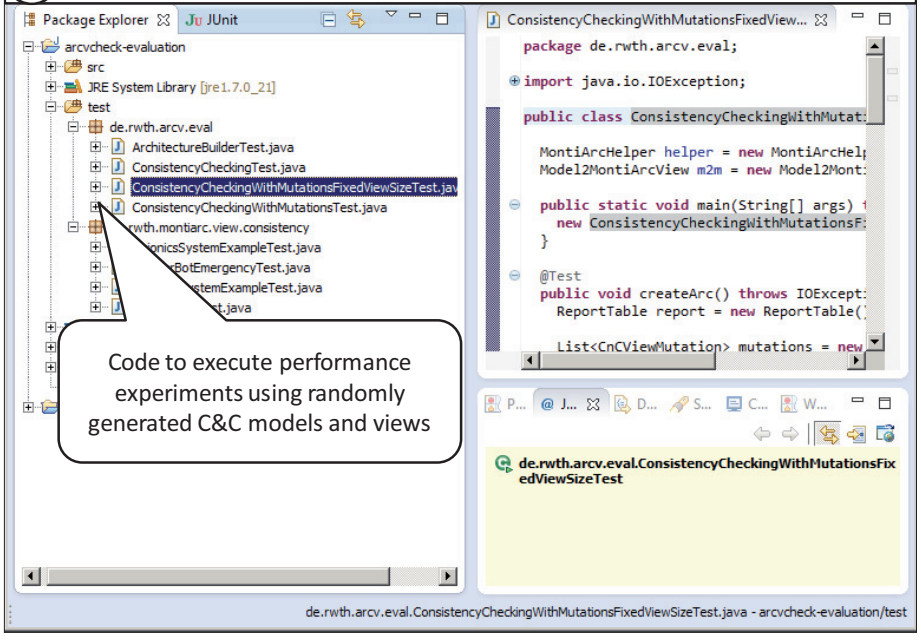
Advanced Evaluation Project Contents

- Code to generate random C&C models and views
- Code to mutate views
- Code to reproduce experiments results

15 C&C Model and C&C View Generator & Mutations



16 Automated Experiments



Appendix D.

How to Use the C&C Views Synthesis Plug-In

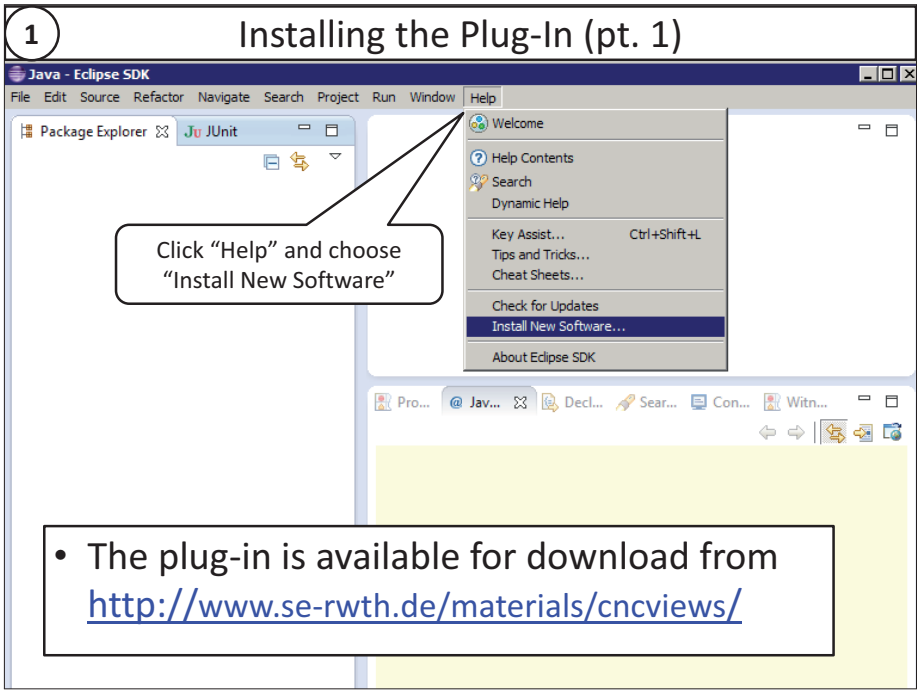
This appendix presents an overview of how to use the C&C views synthesis Eclipse plug-in presented in Chapter 5 for synthesizing C&C models from C&C views specifications.

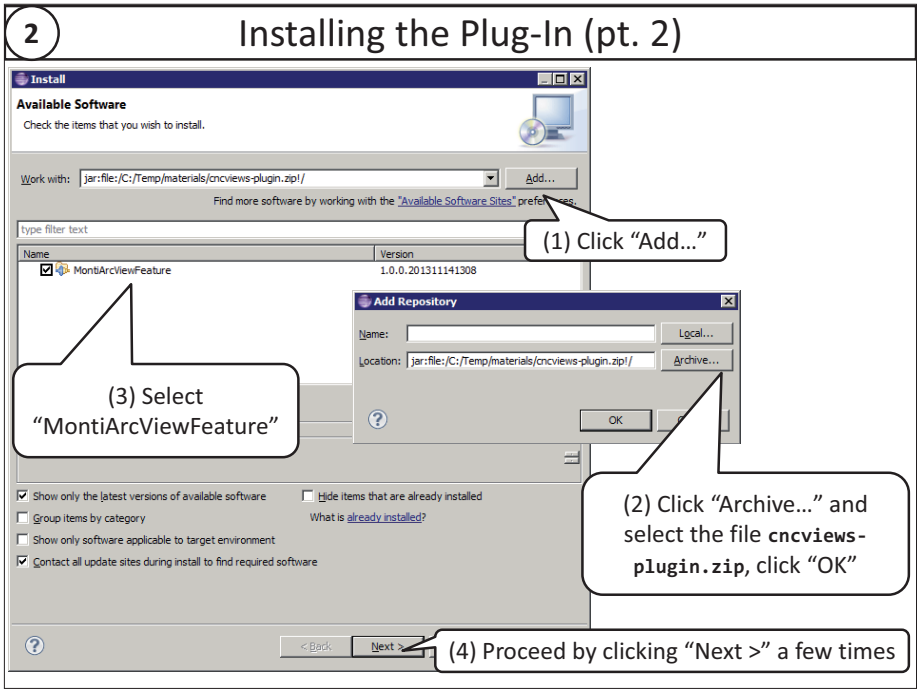
The plug-in comes with various views for the example systems and corresponding C&C views specifications listed in Section 5.6.2.

The plug-in and the evaluation project are available for download from [wwwu]. The first slides on the following pages explain how to install the plug-in in Eclipse and how to import the evaluation project. The plug-in has been tested with the following software:

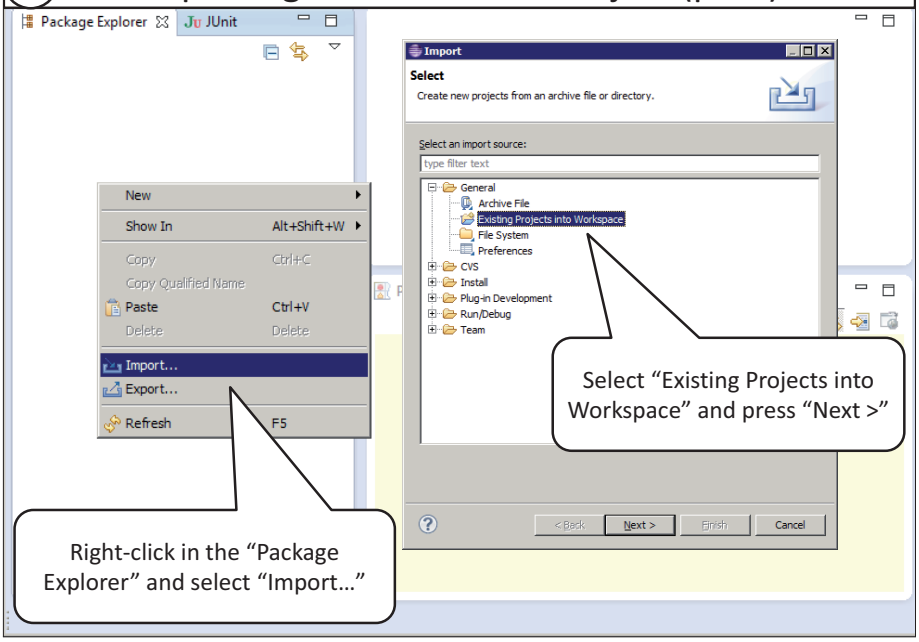
- Eclipse 3.7 (32-bit) and 4.2.2 (32-bit)
- Java SE Development Kit 7, 1.7.0_21 (32-bit)
- Windows 7 (64-bit)

The synthesis functionality of the plug-in is only available on Windows. The synthesis executes a 32-bit Windows executable of the SAT solver MiniSAT version 2.2.0 provided with the plug-in.

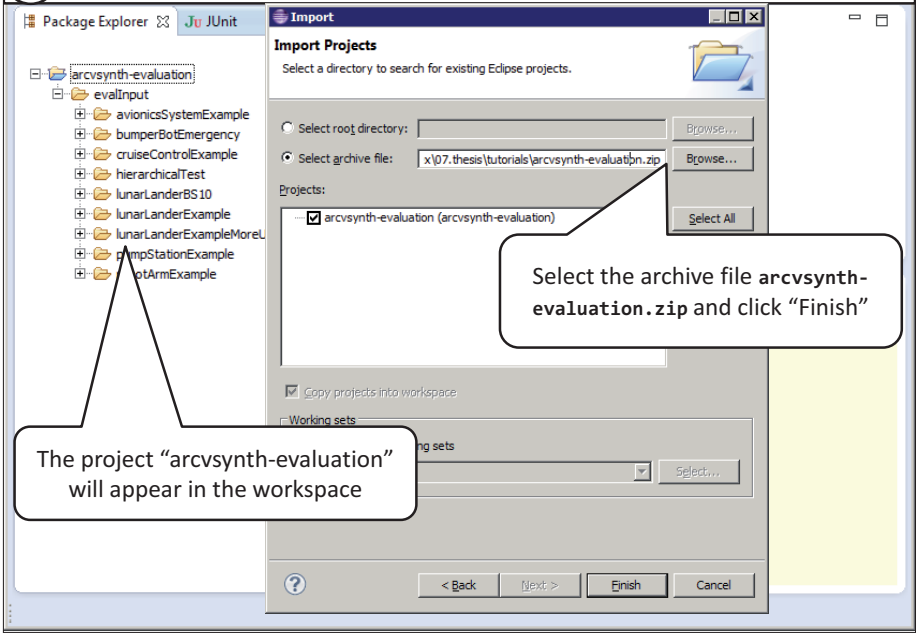

1
Installing the Plug-In (pt. 1)
Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
Welcome
Help Contents
Search
Dynamic Help
Key Assist... Ctrl+Shift+L
Tips and Tricks...
Cheat Sheets...
Check for Updates
Install New Software...
About Eclipse SDK
Click “Help” and choose “Install New Software”
• The plug-in is available for download from http://www.se-rwth.de/materials/cncviews/

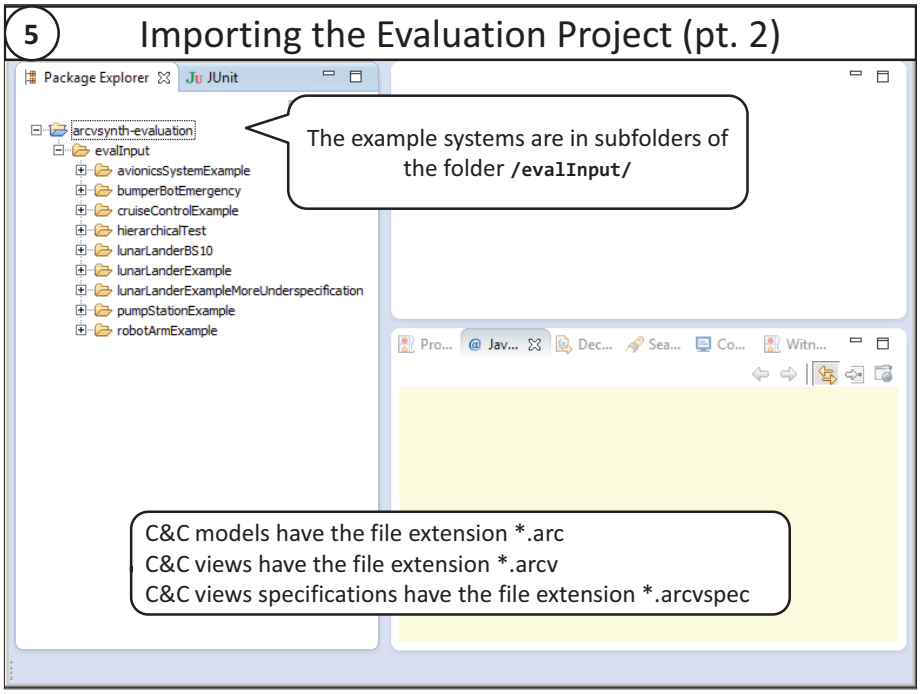

2
Installing the Plug-In (pt. 2)
Install
Available Software
Check the items that you wish to install.
Work with: jar:file:/C:/Temp/materials/cncviews-plugin.zip!/
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
MontiArcViewFeature
1.0.0.201311141308
(1) Click “Add...”
Add Repository
Name:
Local...
Location: jar:file:/C:/Temp/materials/cncviews-plugin.zip!/
Archive...
OK
(3) Select “MontiArcViewFeature”
(2) Click “Archive...” and select the file cncviews-plugin.zip, click “OK”
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
(4) Proceed by clicking “Next >” a few times

3 Importing the Evaluation Project (pt. 1)



4 Importing the Evaluation Project (pt. 2)

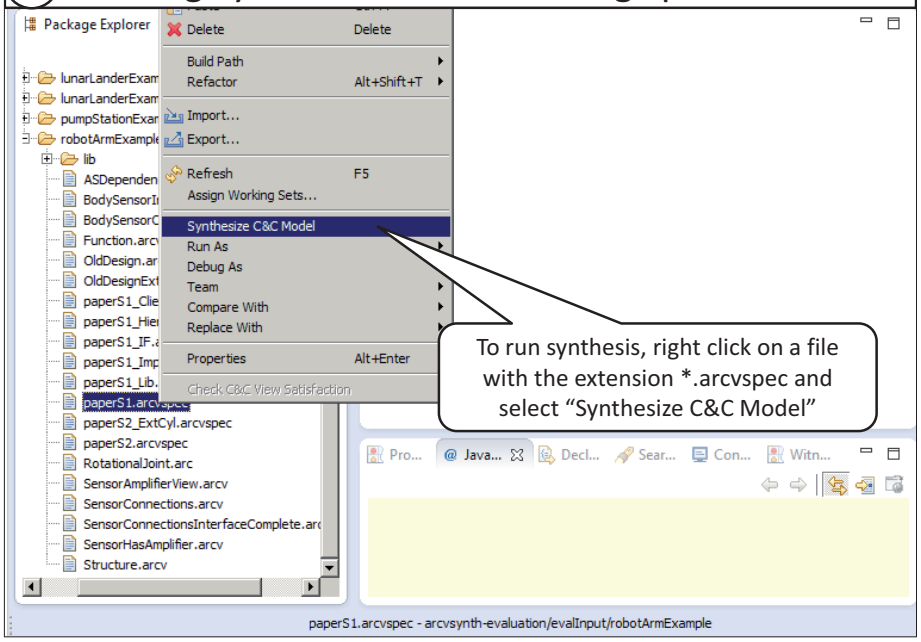




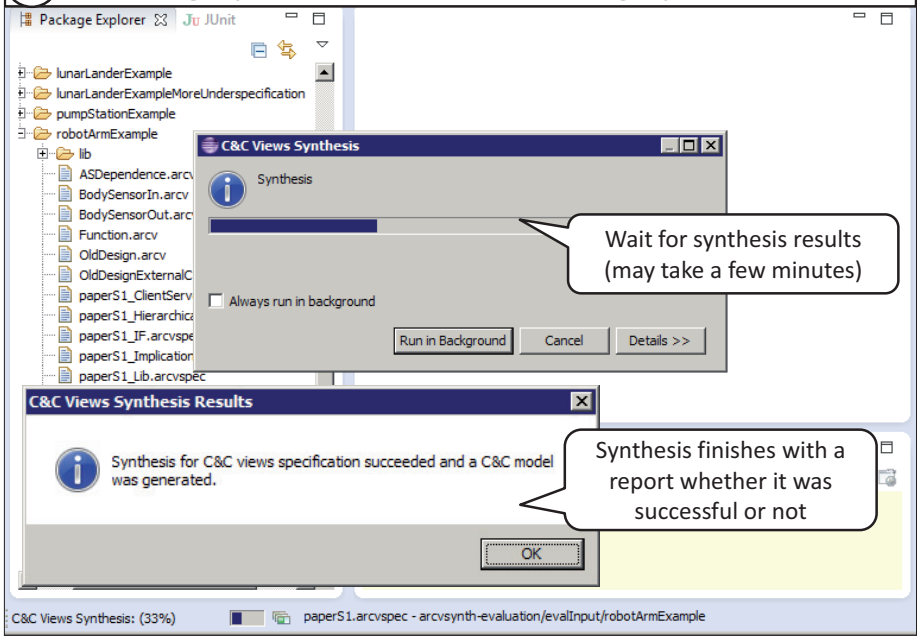
Synthesizing a C&C Model

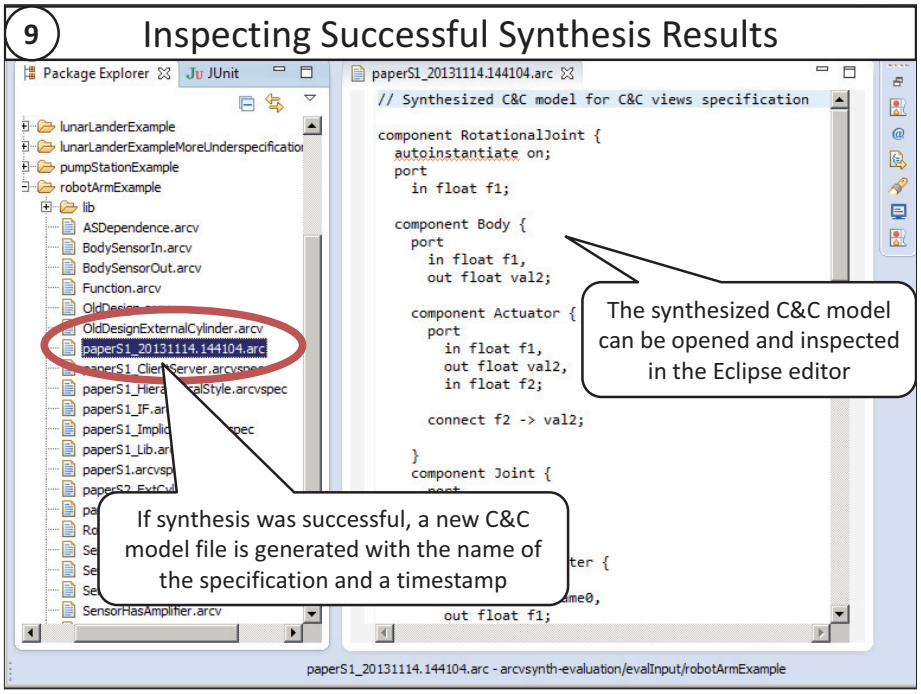
- Running synthesis from an existing specification

7 Running Synthesis with an Existing Specification



8 Running Synthesis with an Existing Specification

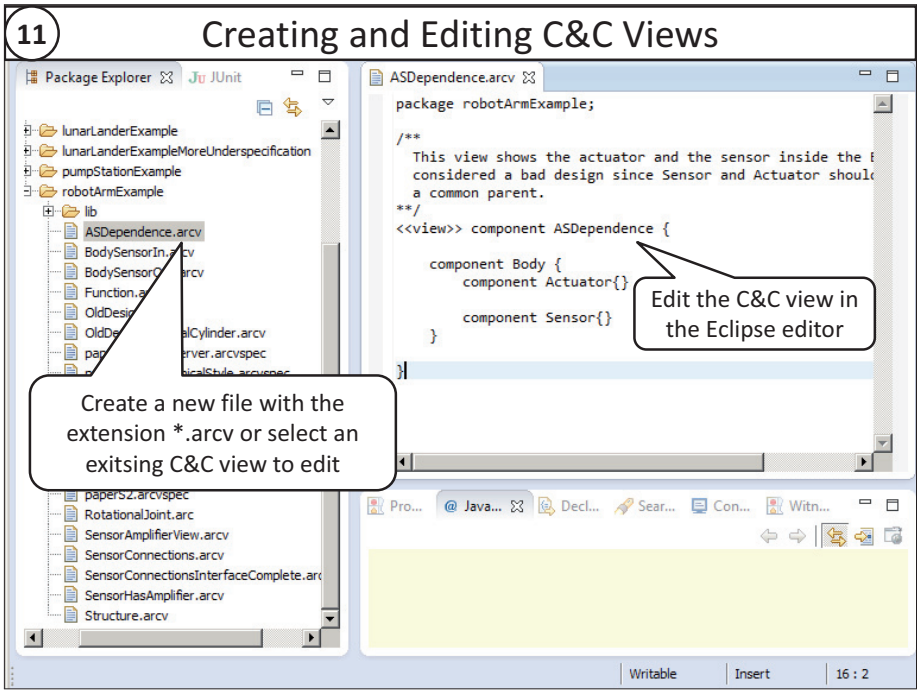


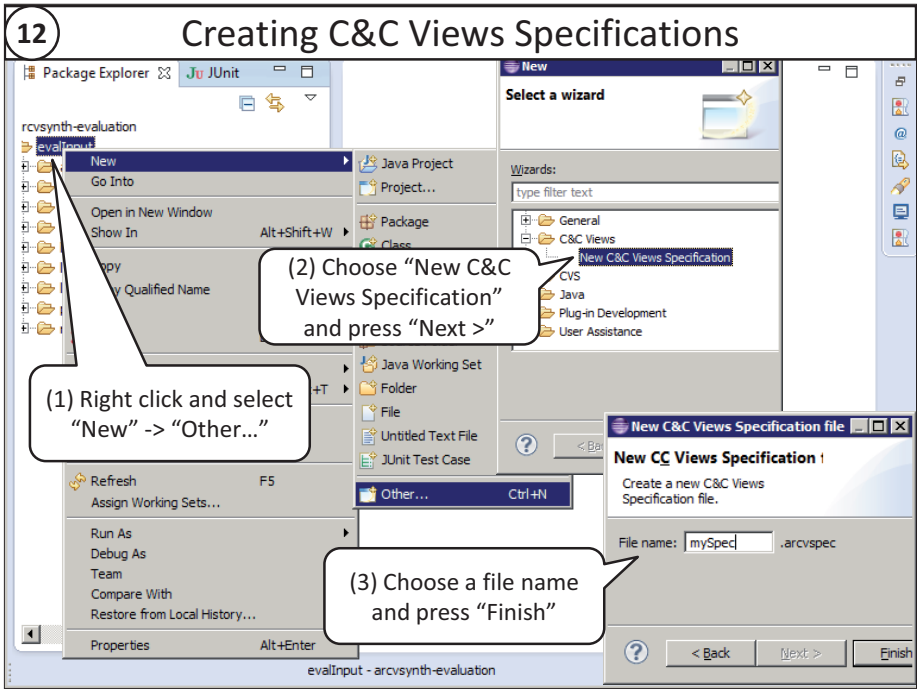


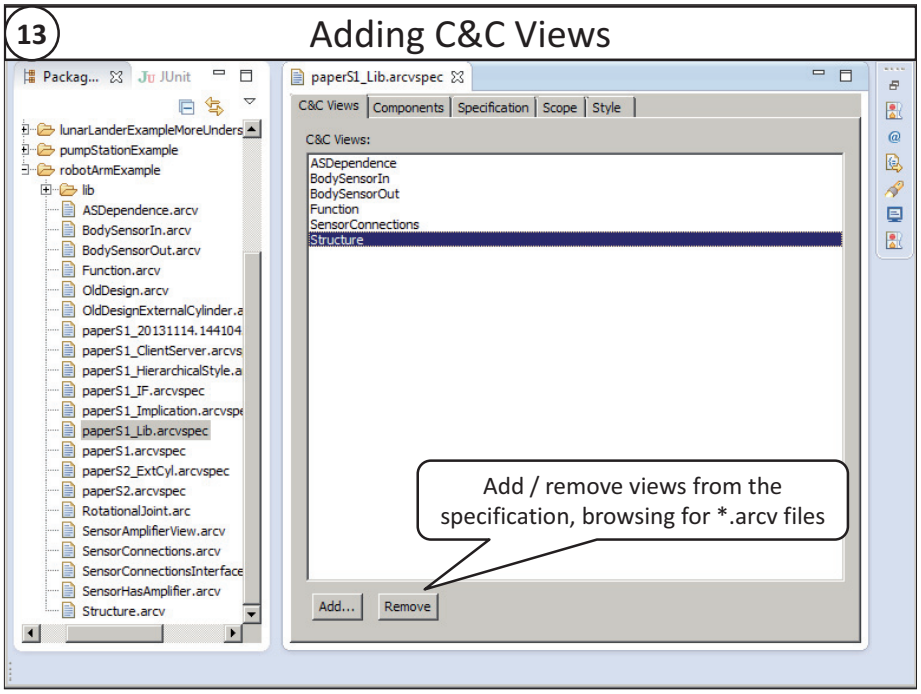
10

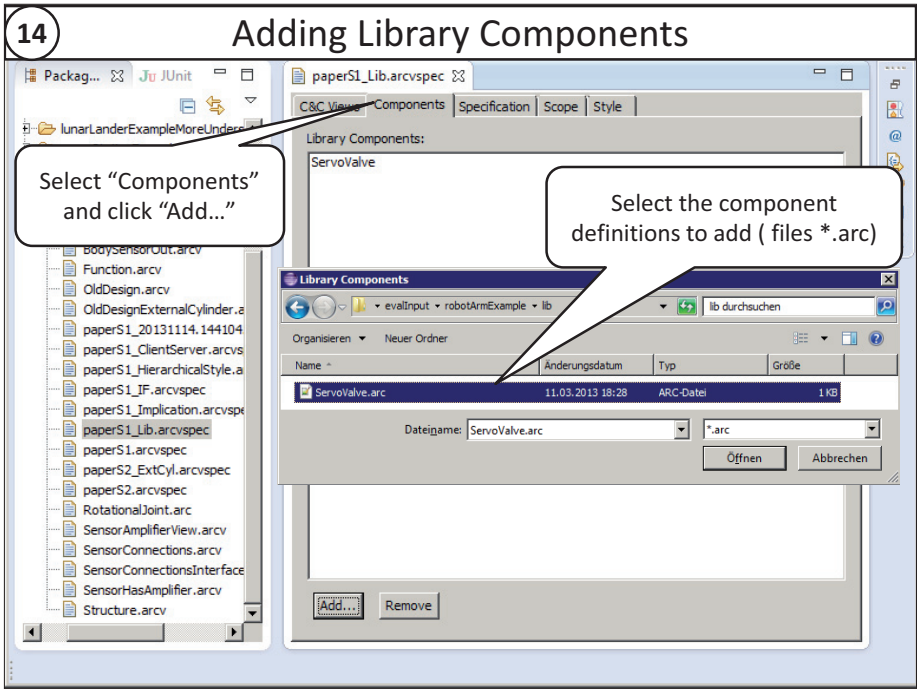
Creating C&C Views Specifications

- Creating or editing C&C views
- Adding C&C views
- Adding library components
- Editing the propositional formula
- Synthesis with architectural styles

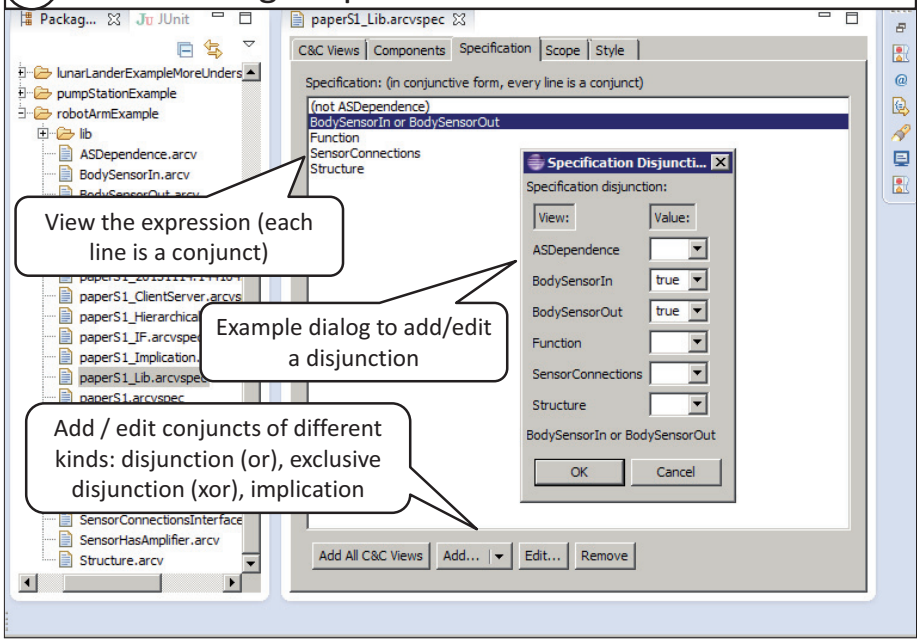

11
Creating and Editing C&C Views
Package Explorer
JUnit
lunarLanderExample
lunarLanderExampleMoreUnderspecification
pumpStationExample
robotArmExample
lib
ASDependence.arcv
Function.a
OldDesi
OldDe
lCylinder.arcv
rver.arcvspec
paperS2.arcvspec
RotationalJoint.arc
SensorAmplifierView.arcv
SensorConnections.arcv
SensorConnectionsInterfaceComplete.arc
SensorHasAmplifier.arcv
Structure.arcv
ASDependence.arcv
package robotArmExample;
/**
This view shows the actuator and the sensor inside the
considered a bad design since Sensor and Actuator shoul
a common parent.
**/
<<view>> component ASDependence {
component Body {
component Actuator{}
component Sensor{}
}
}
Edit the C&C view in the Eclipse editor
Create a new file with the extension *.arcv or select an exitsing C&C view to edit
Pro...
Java...
Decl...
Sear...
Con...
Witn...
Writable
Insert
16 : 2


12
Creating C&C Views Specifications
Package Explorer
JUnit
rcvsynth-evaluation
evalInput
New
Go Into
Open in New Window
Show In
Alt+Shift+W
Java Project
Project...
Package
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Other...
Ctrl+N
Refresh
F5
Assign Working Sets...
Run As
Debug As
Team
Compare With
Restore from Local History...
Properties
Alt+Enter
New
Select a wizard
Wizards:
type filter text
General
C&C Views
New C&C Views Specification
CVS
Java
Plug-in Development
User Assistance
(2) Choose “New C&C Views Specification” and press “Next >”
(1) Right click and select “New” -> “Other...”
New C&C Views Specification file
New C&C Views Specification
Create a new C&C Views Specification file.
File name: mySpec
.arcvspec
(3) Choose a file name and press “Finish”
< Back
Next >
Finish
evalInput - arcvsynth-evaluation

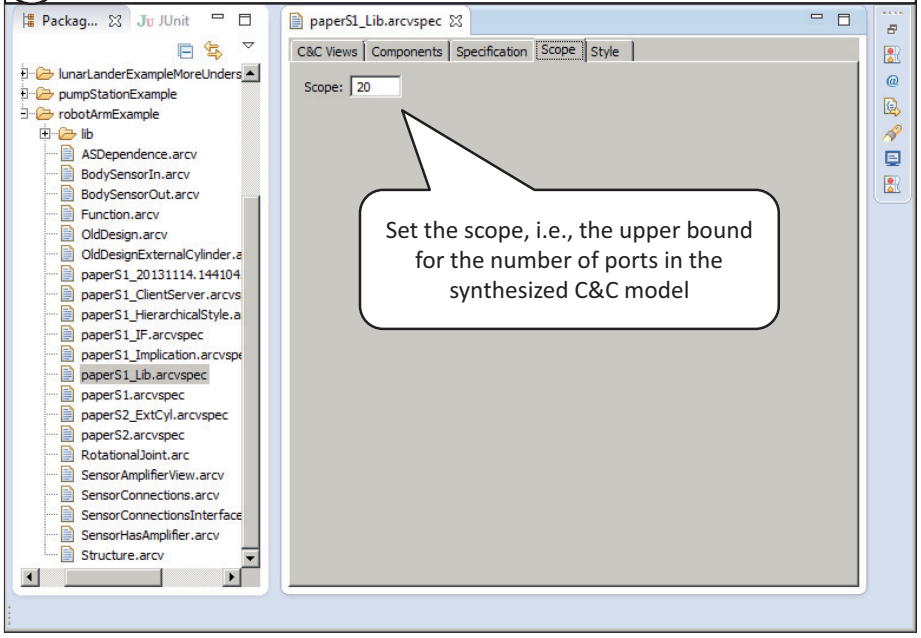

13
Adding C&C Views
Packag...
JUnit
lunarLanderExampleMoreUnders
pumpStationExample
robotArmExample
lib
ASDependence.arcv
BodySensorIn.arcv
BodySensorOut.arcv
Function.arcv
OldDesign.arcv
OldDesignExternalCylinder.a
paperS1_20131114.144104
paperS1_ClientServer.arcvs
paperS1_HierarchicalStyle.a
paperS1_IF.arcvspec
paperS1_Implication.arcvspe
paperS1_Lib.arcvspec
paperS1.arcvspec
paperS2_ExtCyl.arcvspec
paperS2.arcvspec
RotationalJoint.arc
SensorAmplifierView.arcv
SensorConnections.arcv
SensorConnectionsInterface
SensorHasAmplifier.arcv
Structure.arcv
paperS1_Lib.arcvspec
C&C Views
Components
Specification
Scope
Style
C&C Views:
ASDependence
BodySensorIn
BodySensorOut
Function
SensorConnections
Structure
Add / remove views from the specification, browsing for *.arcv files
Add...
Remove

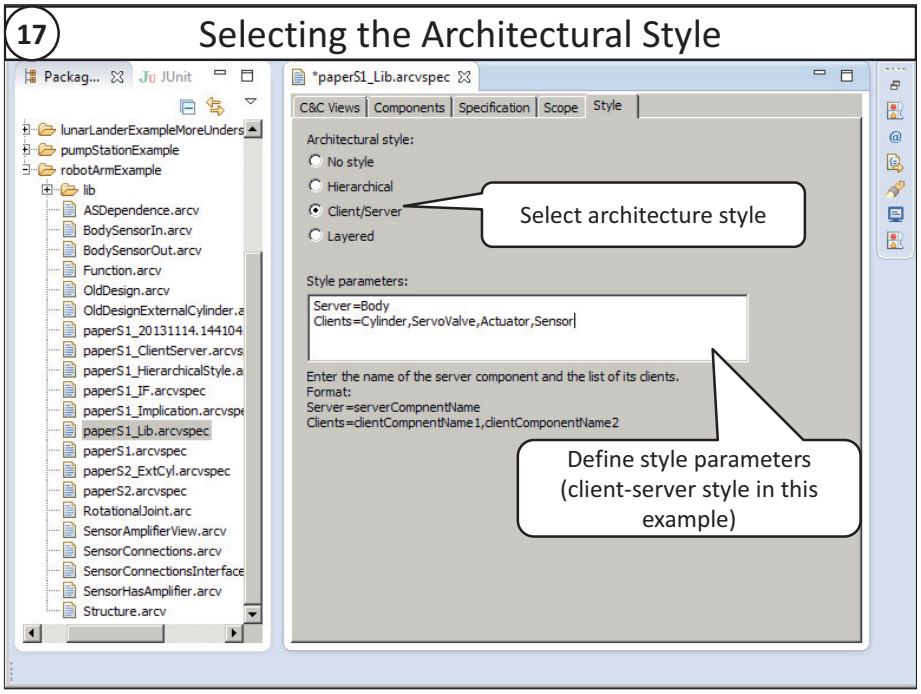

14
Adding Library Components
Packag...
JUnit
paperS1_Lib.arcvspec
C&C Views
Components
Specification
Scope
Style
Library Components:
ServoValve
Select “Components” and click “Add...”
Select the component definitions to add (files *.arc)
Library Components
evalInput
robotArmExample
lib
lib durchsuchen
Organisieren
Neuer Ordner
Name
Änderungsdatum
Typ
Größe
ServoValve.arc
11.03.2013 18:28
ARC-Datei
1 KB
Dateiname:
ServoValve.arc
*.arc
Öffnen
Abbrechen
Add...
Remove

15 Editing Propositional Views Formula



16 Setting the Scope for Synthesis





18

Architectural Style Parameters

- No parameters are required for **`Hierarchical`** style
- For **`Client/Server`**, enter the name of the server component and the list of its clients

  server=serverCompName

  clients=clientCompName1,clientCompName2
- For **`Layered`** style, enter the list of layers, one layer per line, in order. For each layer add the list of its subcomponents (if known)

  layerName1=comp1,comp2

  layerName2=comp3

Appendix E.

How to Use the MontiArcAutomaton Verification Implementation

This appendix explains how to use the MontiArcAutomaton verification implementation. The MontiArcAutomaton verification framework does not require the installation as a plug-in. The necessary MontiArcAutomaton parsers and generators are resolved from a public repository by the Maven *software project management and comprehension tool.*[1] The use of Maven requires the installation of the Eclipse m2e Maven plug-in and a Java SE Development Kit.

We provide the Eclipse project `bumperbot-verification` which contains the MontiArcAutomaton models and specification suites of the bumper bot presented in Section 7.1 and in Section 7.5. Additional MontiArcAutomaton models and specification suites described in Section 7.6 are provided in the Eclipse project `maa-verification`.

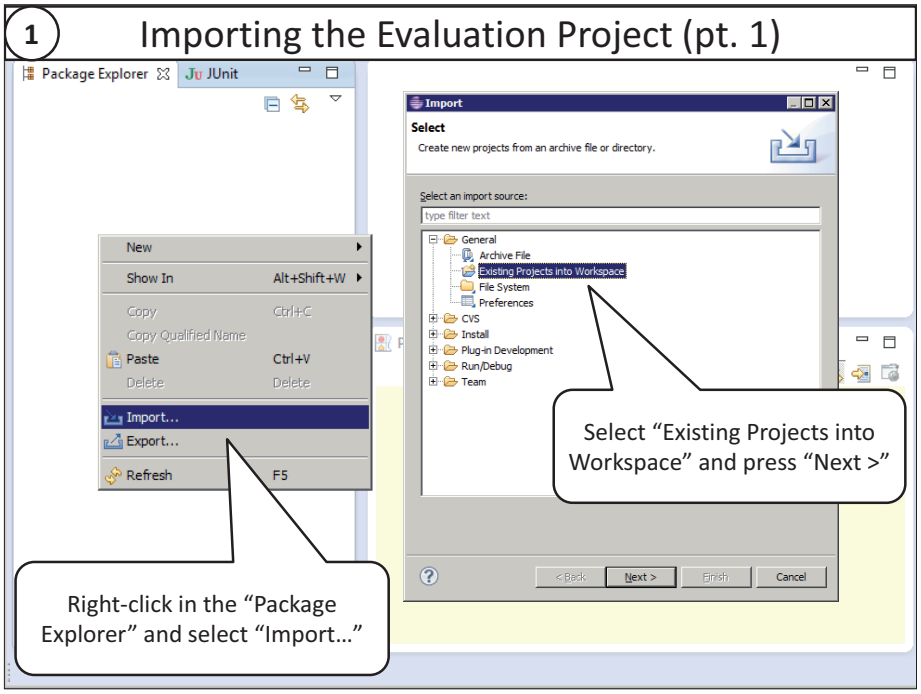
Both evaluation projects are available for download from [wwws]. The slides on the following pages explain how to import the `bumperbot-verification` project and how to verify specification suites. The plug-in has been tested with the following software:

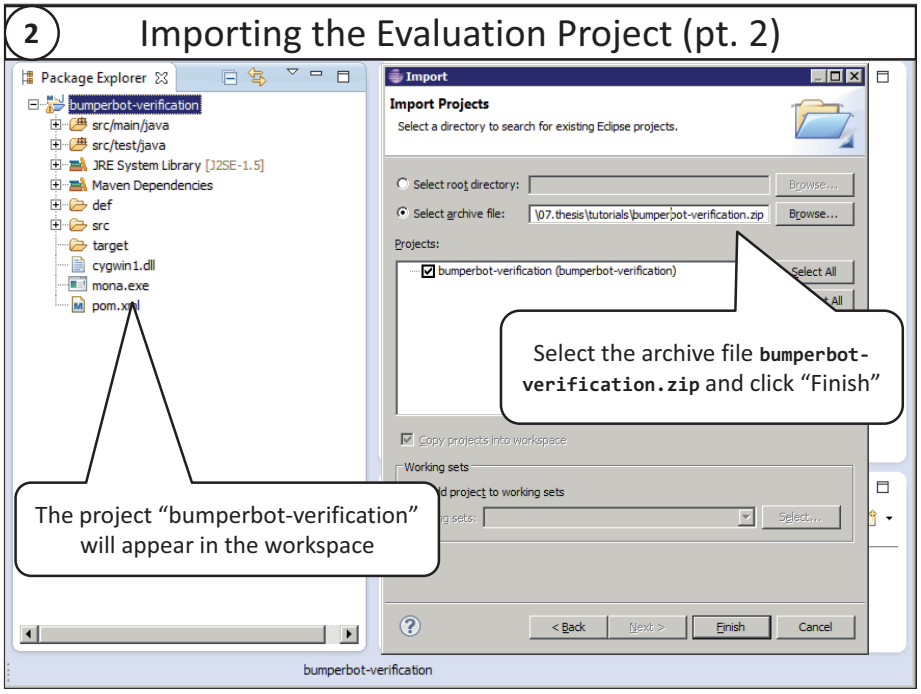
- Eclipse 3.7 (32-bit) and 4.2.2 (32-bit), with the installed plug-in:
  - Maven plug-in m2e – Maven Integration for Eclipse 1.4.0[2]
- Java SE Development Kit 7, 1.7.0_21 (32-bit)
- Windows 7 (64-bit)

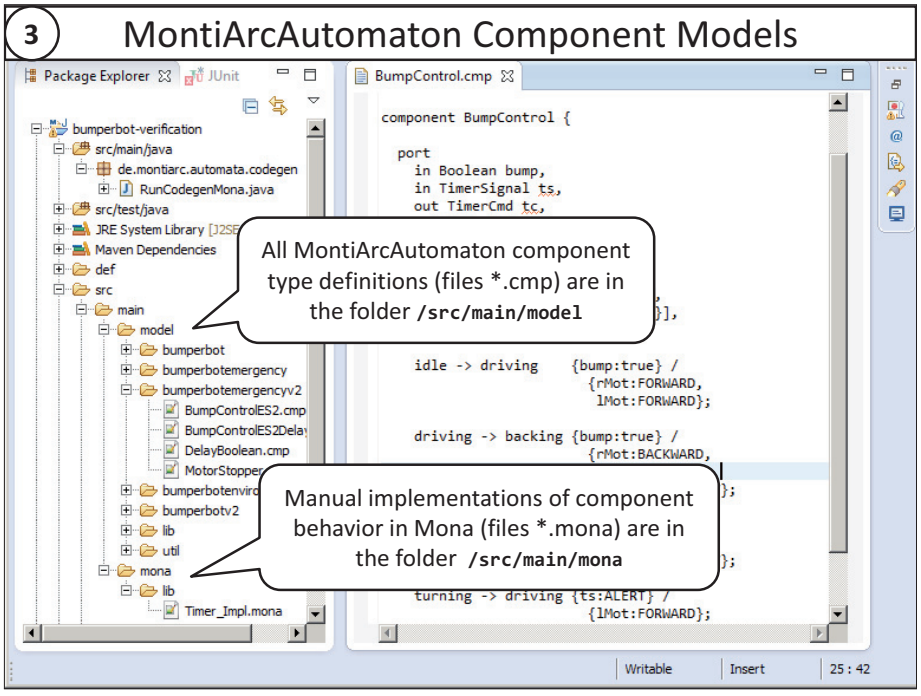
The verification functionality of the plug-in is only available on Windows. The verification executes a 32-bit Windows executable of Mona version 1.4-13 provided with the example projects.

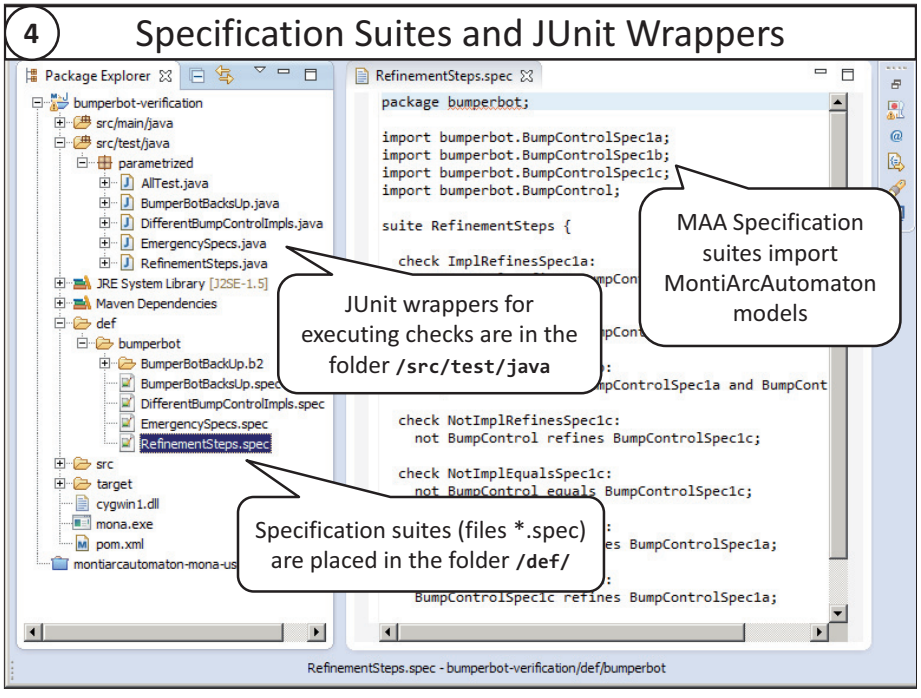
[1] `http://maven.apache.org/` (accessed 11/13)

[2] Available from m2e Update Site – `http://download.eclipse.org/technology/m2e/releases` (accessed 11/13)


1
Importing the Evaluation Project (pt. 1)
Package Explorer
JUnit
New
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Import...
Export...
Refresh F5
Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
CVS
Install
Plug-in Development
Run/Debug
Team
< Back
Next >
Finish
Cancel
Right-click in the "Package Explorer" and select "Import..."
Select "Existing Projects into Workspace" and press "Next >"

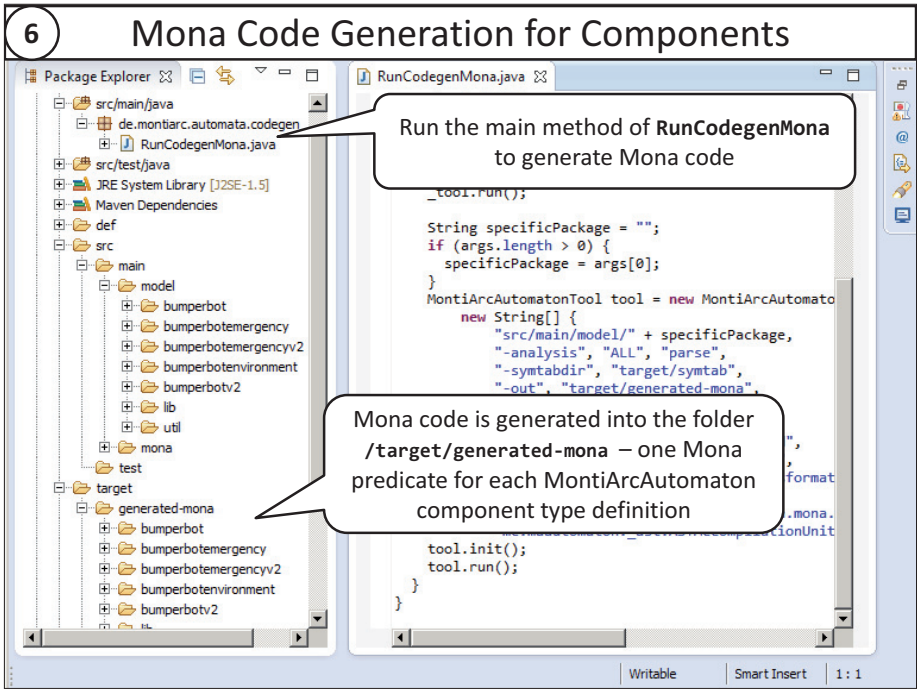

2
Importing the Evaluation Project (pt. 2)
Package Explorer
bumperbot-verification
src/main/java
src/test/java
JRE System Library [J2SE-1.5]
Maven Dependencies
def
src
target
cygwin1.dll
mona.exe
pom.xml
Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Select archive file:
\07.thesis\tutorials\bumperbot-verification.zip
Browse...
Projects:
bumperbot-verification (bumperbot-verification)
Select All
Copy projects into workspace
Working sets
< Back
Next >
Finish
Cancel
bumperbot-verification
The project "bumperbot-verification" will appear in the workspace
Select the archive file bumperbot-verification.zip and click "Finish"

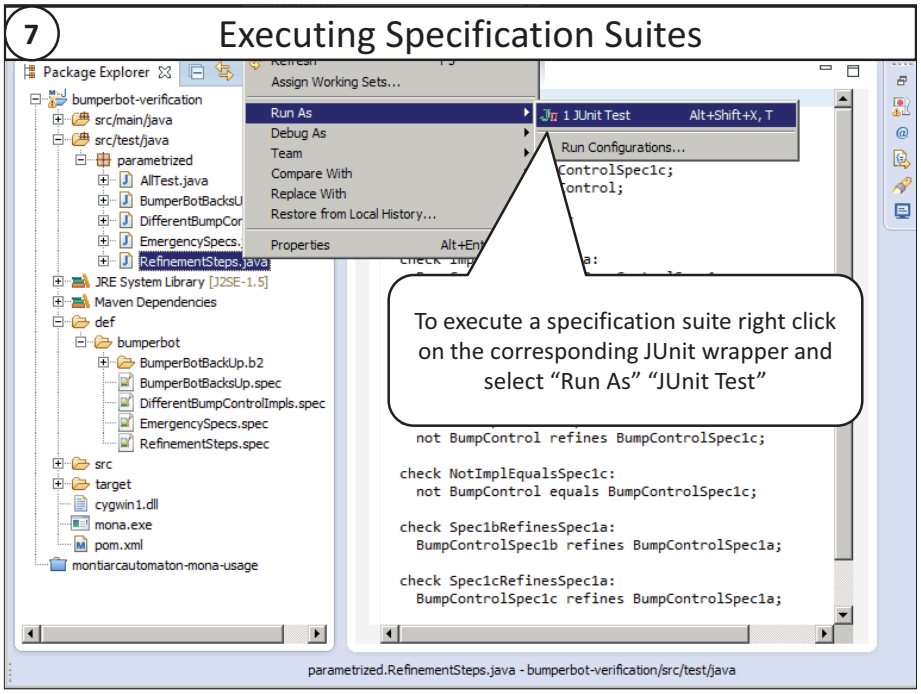

3
MontiArcAutomaton Component Models
Package Explorer
JUnit
BumpControl.cmp
bumperbot-verification
src/main/java
de.montiarc.automata.codegen
RunCodegenMona.java
src/test/java
JRE System Library [J2SE
Maven Dependencies
def
src
main
model
bumperbot
bumperbotemergency
bumperbotemergencyv2
BumpControlES2.cmp
BumpControlES2Dela
DelayBoolean.cmp
MotorStopper
bumperbotenviro
bumperbotv2
lib
util
mona
lib
Timer_Impl.mona
component BumpControl {
port
in Boolean bump,
in TimerSignal ts,
out TimerCmd tc,
idle -> driving {bump:true} /
{rMot:FORWARD,
lMot:FORWARD};
driving -> backing {bump:true} /
{rMot:BACKWARD,
turning -> driving {ts:ALERT} /
{lMot:FORWARD};
All MontiArcAutomaton component type definitions (files *.cmp) are in the folder /src/main/model
Manual implementations of component behavior in Mona (files *.mona) are in the folder /src/main/mona
Writable
Insert
25 : 42

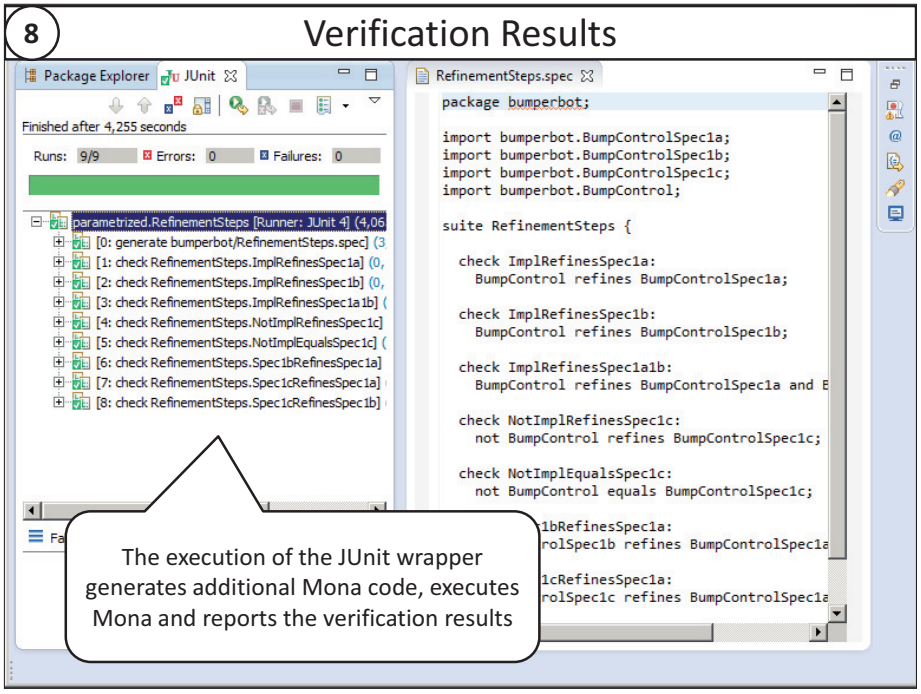

4
Specification Suites and JUnit Wrappers
Package Explorer
RefinementSteps.spec
bumperbot-verification
src/main/java
src/test/java
parametrized
AllTest.java
BumperBotBacksUp.java
DifferentBumpControlImpls.java
EmergencySpecs.java
RefinementSteps.java
JRE System Library [J2SE-1.5]
Maven Dependencies
def
bumperbot
BumperBotBackUp.b2
BumperBotBacksUp.spec
DifferentBumpControlImpls.spec
EmergencySpecs.spec
RefinementSteps.spec
src
target
cygwin1.dll
mona.exe
pom.xml
montiarcautomaton-mona-us
package bumperbot;
import bumperbot.BumpControlSpec1a;
import bumperbot.BumpControlSpec1b;
import bumperbot.BumpControlSpec1c;
import bumperbot.BumpControl;
suite RefinementSteps {
check ImplRefinesSpec1a:
check NotImplRefinesSpec1c:
not BumpControl refines BumpControlSpec1c;
check NotImplEqualsSpec1c:
not BumpControl equals BumpControlSpec1c;
BumpControlSpec1c refines BumpControlSpec1a;
MAA Specification suites import MontiArcAutomaton models
JUnit wrappers for executing checks are in the folder /src/test/java
Specification suites (files *.spec) are placed in the folder /def/
RefinementSteps.spec - bumperbot-verification/def/bumperbot

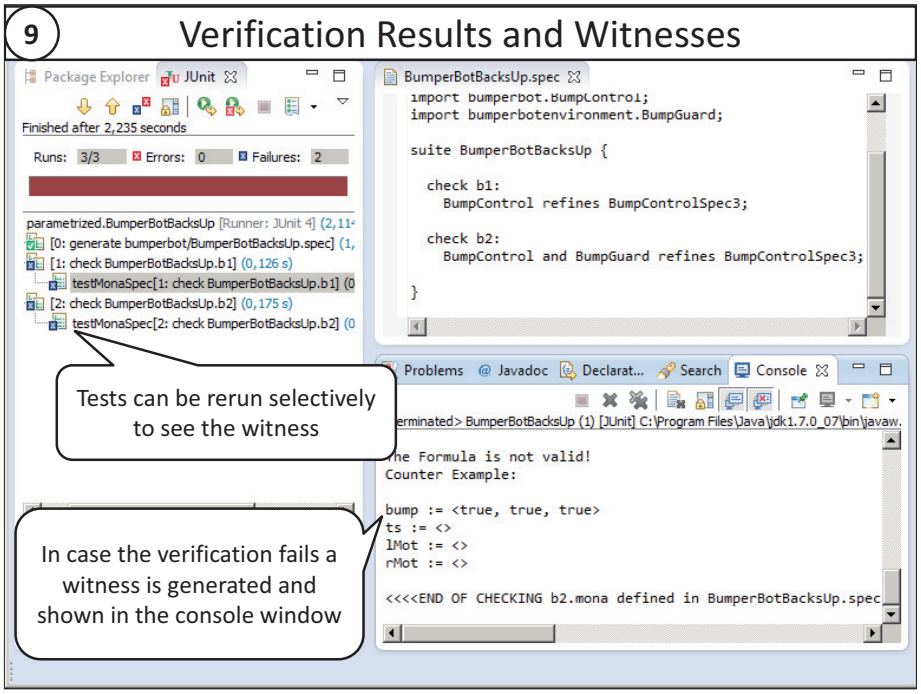
Running Existing Specification Suites

- Generating Mona code for components
- Executing JUnit wrappers for specification suites
- Result of verification




7
Executing Specification Suites
Package Explorer
bumperbot-verification
src/main/java
src/test/java
parametrized
AllTest.java
DifferentBumpCor
EmergencySpecs.
RefinementSteps.java
JRE System Library [J2SE-1.5]
Maven Dependencies
def
bumperbot
BumperBotBackUp.b2
BumperBotBacksUp.spec
DifferentBumpControlImpls.spec
EmergencySpecs.spec
RefinementSteps.spec
src
target
cygwin1.dll
mona.exe
pom.xml
montiarcautomaton-mona-usage
Assign Working Sets...
Run As
Debug As
Team
Compare With
Replace With
Restore from Local History...
Properties
1 JUnit Test
Alt+Shift+X, T
Run Configurations...
not BumpControl refines BumpControlSpec1c;
check NotImplEqualsSpec1c:
not BumpControl equals BumpControlSpec1c;
check Spec1bRefinesSpec1a:
BumpControlSpec1b refines BumpControlSpec1a;
check Spec1cRefinesSpec1a:
BumpControlSpec1c refines BumpControlSpec1a;
To execute a specification suite right click on the corresponding JUnit wrapper and select “Run As” “JUnit Test”
parametrized.RefinementSteps.java - bumperbot-verification/src/test/java

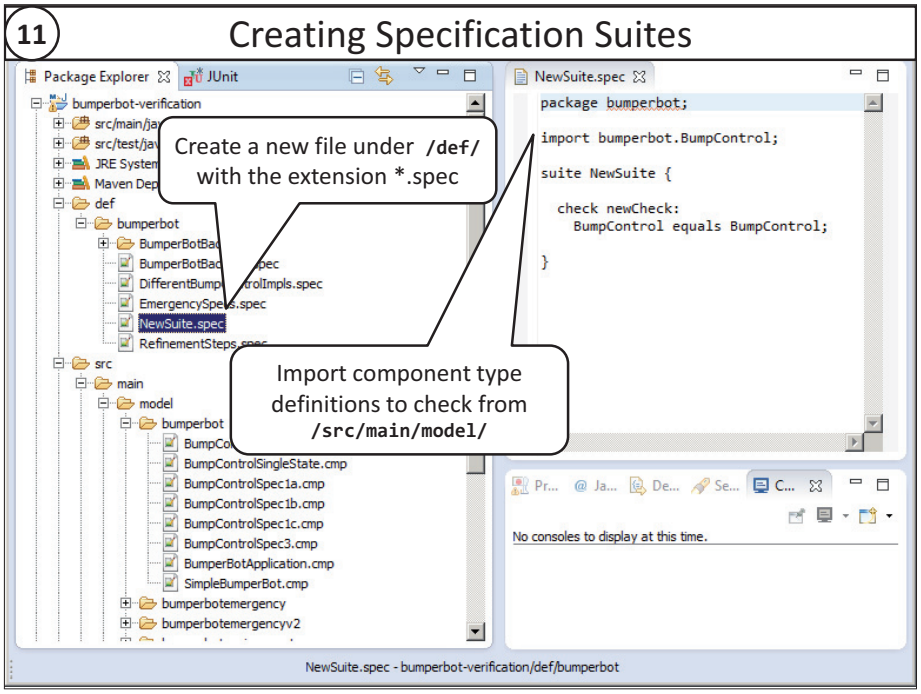

8
Verification Results
Package Explorer
JUnit
Finished after 4,255 seconds
Runs: 9/9
Errors: 0
Failures: 0
parametrized.RefinementSteps [Runner: JUnit 4] (4,06
[0: generate bumperbot/RefinementSteps.spec] (3
[1: check RefinementSteps.ImplRefinesSpec1a] (0,
[2: check RefinementSteps.ImplRefinesSpec1b] (0,
[3: check RefinementSteps.ImplRefinesSpec1a1b] (
[4: check RefinementSteps.NotImplRefinesSpec1c]
[5: check RefinementSteps.NotImplEqualsSpec1c] (
[6: check RefinementSteps.Spec1bRefinesSpec1a]
[7: check RefinementSteps.Spec1cRefinesSpec1a]
[8: check RefinementSteps.Spec1cRefinesSpec1b]
RefinementSteps.spec
package bumperbot;
import bumperbot.BumpControlSpec1a;
import bumperbot.BumpControlSpec1b;
import bumperbot.BumpControlSpec1c;
import bumperbot.BumpControl;
suite RefinementSteps {
check ImplRefinesSpec1a:
BumpControl refines BumpControlSpec1a;
check ImplRefinesSpec1b:
BumpControl refines BumpControlSpec1b;
check ImplRefinesSpec1a1b:
BumpControl refines BumpControlSpec1a and E
check NotImplRefinesSpec1c:
not BumpControl refines BumpControlSpec1c;
check NotImplEqualsSpec1c:
not BumpControl equals BumpControlSpec1c;
The execution of the JUnit wrapper generates additional Mona code, executes Mona and reports the verification results

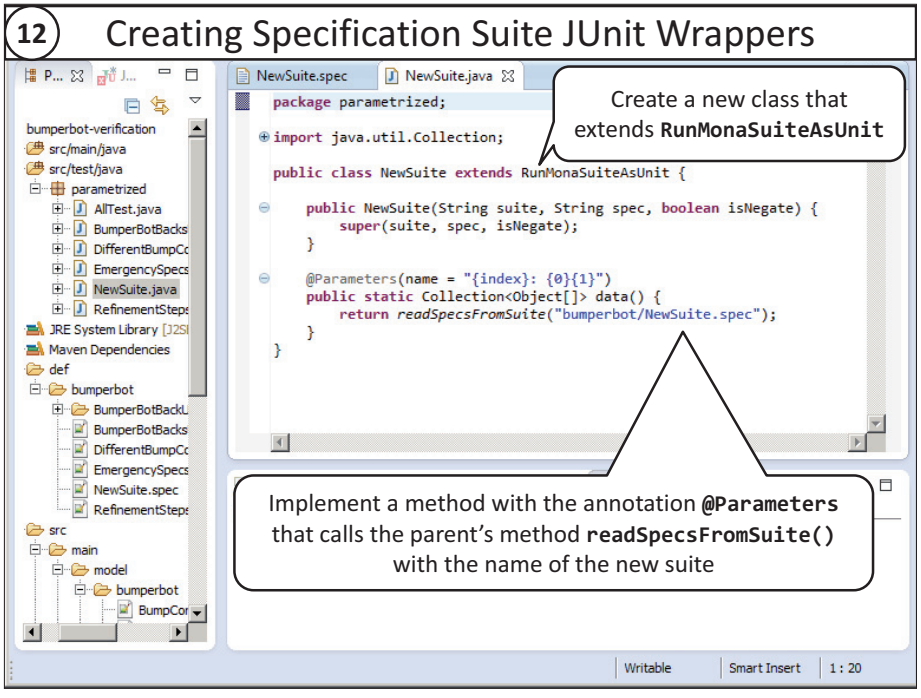


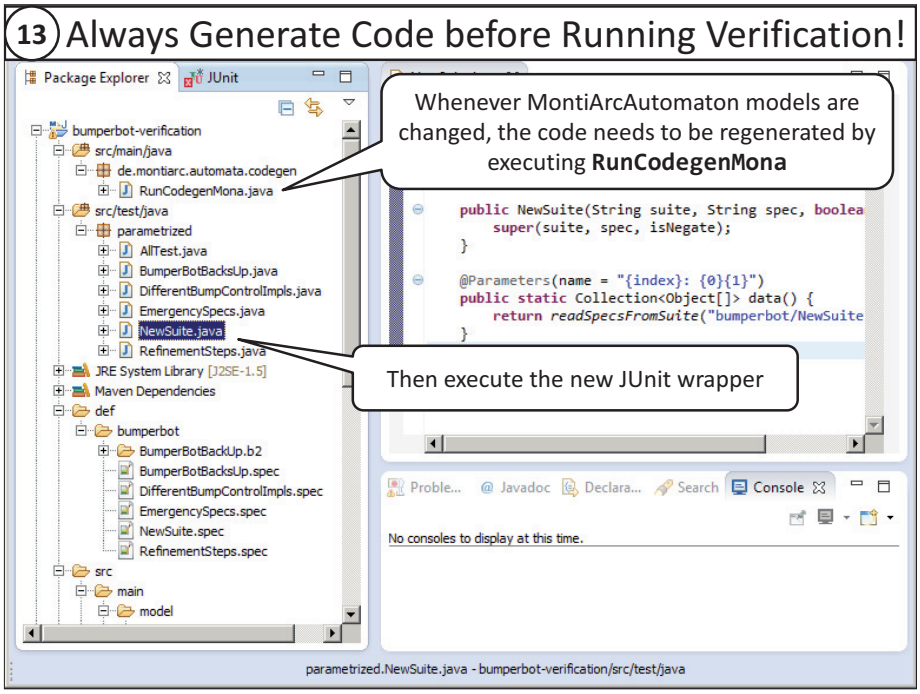
10

Creating Specification Suites

- Creating specification suites
- Setting up the JUnit wrapper


11
Creating Specification Suites
Package Explorer
JUnit
bumperbot-verification
def
bumperbot
NewSuite.spec
src
main
model
BumpControlSingleState.cmp
BumpControlSpec1a.cmp
BumpControlSpec1b.cmp
BumpControlSpec1c.cmp
BumpControlSpec3.cmp
BumperBotApplication.cmp
SimpleBumperBot.cmp
bumperbotemergency
bumperbotemergencyv2
Create a new file under /def/ with the extension *.spec
Import component type definitions to check from /src/main/model/
package bumperbot;
import bumperbot.BumpControl;
suite NewSuite {
check newCheck:
BumpControl equals BumpControl;
}
No consoles to display at this time.
NewSuite.spec - bumperbot-verification/def/bumperbot


12
Creating Specification Suite JUnit Wrappers
NewSuite.spec
NewSuite.java
bumperbot-verification
src/main/java
src/test/java
parametrized
AllTest.java
NewSuite.java
JRE System Library
Maven Dependencies
def
bumperbot
src
main
model
bumperbot
Create a new class that extends RunMonaSuiteAsUnit
package parametrized;
import java.util.Collection;
public class NewSuite extends RunMonaSuiteAsUnit {
public NewSuite(String suite, String spec, boolean isNegate) {
super(suite, spec, isNegate);
}
@Parameters(name = "{index}: {0}{1}")
public static Collection<Object[]> data() {
return readSpecsFromSuite("bumperbot/NewSuite.spec");
}
}
Implement a method with the annotation @Parameters that calls the parent's method readSpecsFromSuite() with the name of the new suite
Writable
Smart Insert
1 : 20


13 Always Generate Code before Running Verification!
Package Explorer
JUnit
bumperbot-verification
src/main/java
de.montiarc.automata.codegen
RunCodegenMona.java
src/test/java
parametrized
AllTest.java
BumperBotBacksUp.java
DifferentBumpControlImpls.java
EmergencySpecs.java
NewSuite.java
RefinementSteps.java
JRE System Library [J2SE-1.5]
Maven Dependencies
def
bumperbot
BumperBotBackUp.b2
BumperBotBacksUp.spec
DifferentBumpControlImpls.spec
EmergencySpecs.spec
NewSuite.spec
RefinementSteps.spec
src
main
model
Whenever MontiArcAutomaton models are changed, the code needs to be regenerated by executing RunCodegenMona
public NewSuite(String suite, String spec, boolea
super(suite, spec, isNegate);
}
@Parameters(name = "{index}: {0}{1}")
public static Collection<Object[]> data() {
return readSpecsFromSuite("bumperbot/NewSuite
}
Then execute the new JUnit wrapper
Proble...
Javadoc
Declara...
Search
Console
No consoles to display at this time.
parametrized.NewSuite.java - bumperbot-verification/src/test/java

Appendix F.

How to Use the MontiArcAutomaton Java Code Generator

This appendix explains how to use the MontiArcAutomaton Java code generator. The MontiArcAutomaton Java code generator does not require the installation as a plug-in. The necessary MontiArcAutomaton and UML/P parsers and generators are resolved from a public repository by the Maven *software project management and comprehension tool.*[1] The use of Maven requires the installation of the Eclipse m2e Maven plug-in and a Java SE Development Kit.

We provide the Eclipse project `bumperbot-codegen` which contains the MontiArcAutomaton models of the bumper bot presented in Section 6.1.

The evaluation project is available for download from [wwwr]. The slides on the following pages explain how to import the `bumperbot-codegen` project and how to generate code. The code generator has been tested with the following software:

- Eclipse 3.7 (32-bit) and 4.2.2 (32-bit), with the installed plug-ins:
  - Maven plug-in m2e – Maven Integration for Eclipse 1.4.0[2]
  - leJOS NXJ plug-in 0.9.0[3]
- Java SE Development Kit 7, 1.7.0_21 (32-bit)
- leJOS 0.9.1beta-3 (32-bit)[4]
- Windows 7 (64-bit)

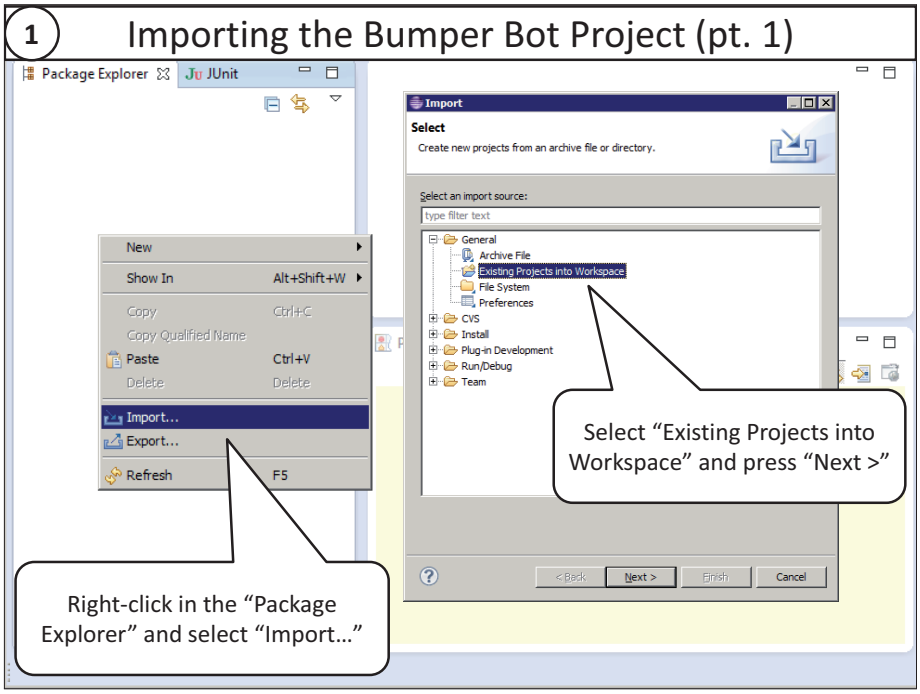
The deployment of the generated code to a Lego NXT robot requires the installation of the leJOS framework and the USB driver included in the leJOS installation package. The code generation itself does not require the installation of leJOS or the leJOS plug-in.

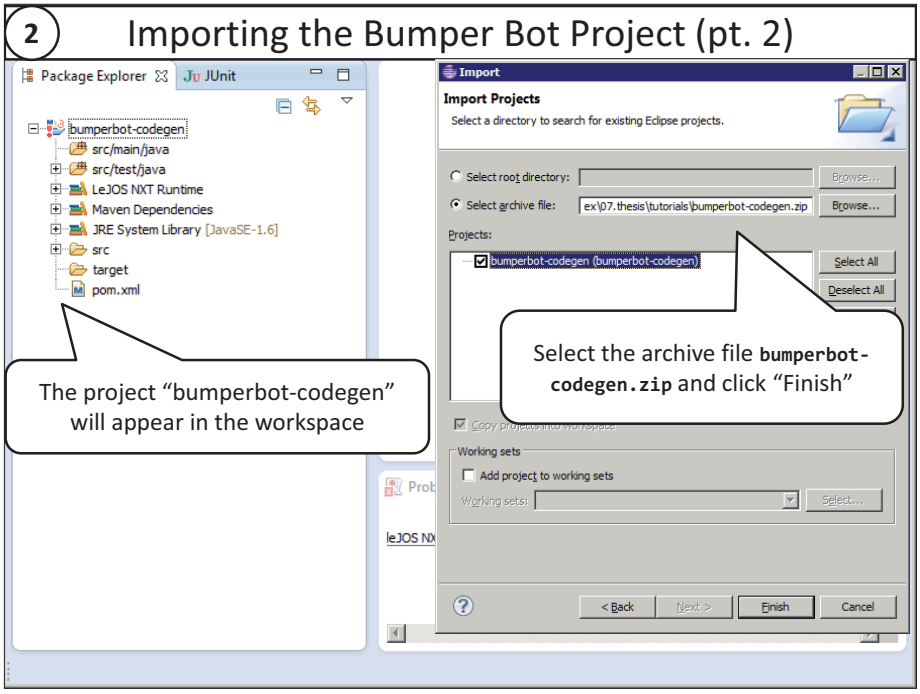
[1] `http://maven.apache.org/` (accessed 11/13)

[2] Available from m2e Update Site – `http://download.eclipse.org/technology/m2e/releases` (accessed 11/13)

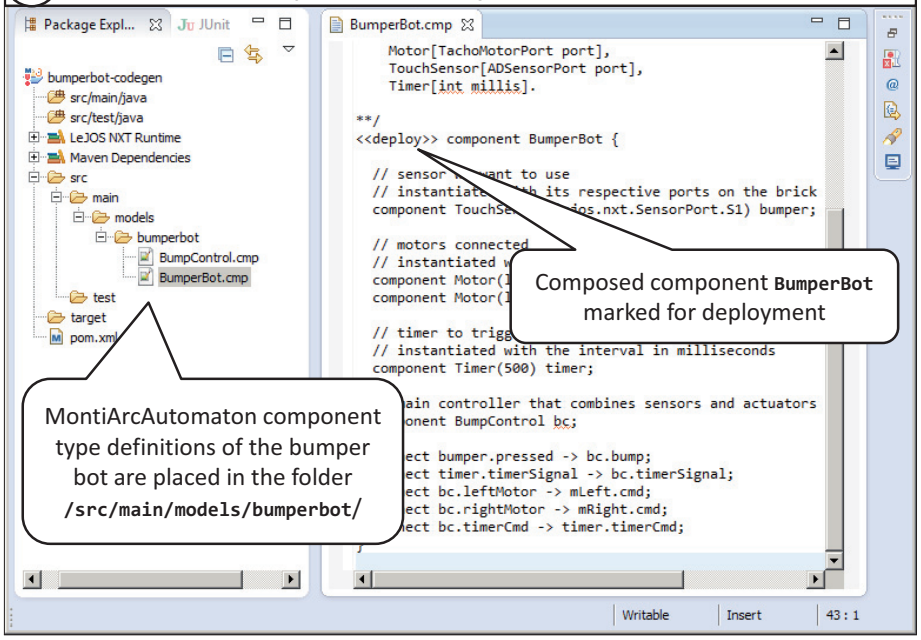
[3] Available from LeJOS Update Site – `http://www.lejos.org/tools/eclipse/plugin/nxj/` (accessed 11/13)

[4] Available from `http://sourceforge.net/projects/lejos/files/lejos-NXJ/0.9.1beta/` (accessed 11/13)


1
Importing the Bumper Bot Project (pt. 1)
Package Explorer
JUnit
New
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Import...
Export...
Refresh F5
Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
CVS
Install
Plug-in Development
Run/Debug
Team
< Back
Next >
Finish
Cancel
Right-click in the "Package Explorer" and select "Import..."
Select "Existing Projects into Workspace" and press "Next >"


2
Importing the Bumper Bot Project (pt. 2)
Package Explorer
JUnit
bumperbot-codegen
src/main/java
src/test/java
LeJOS NXT Runtime
Maven Dependencies
JRE System Library [JavaSE-1.6]
src
target
pom.xml
Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Select archive file:
ex\07.thesis\tutorials\bumperbot-codegen.zip
Browse...
Projects:
bumperbot-codegen (bumperbot-codegen)
Select All
Deselect All
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel
The project "bumperbot-codegen" will appear in the workspace
Select the archive file bumperbot-codegen.zip and click "Finish"

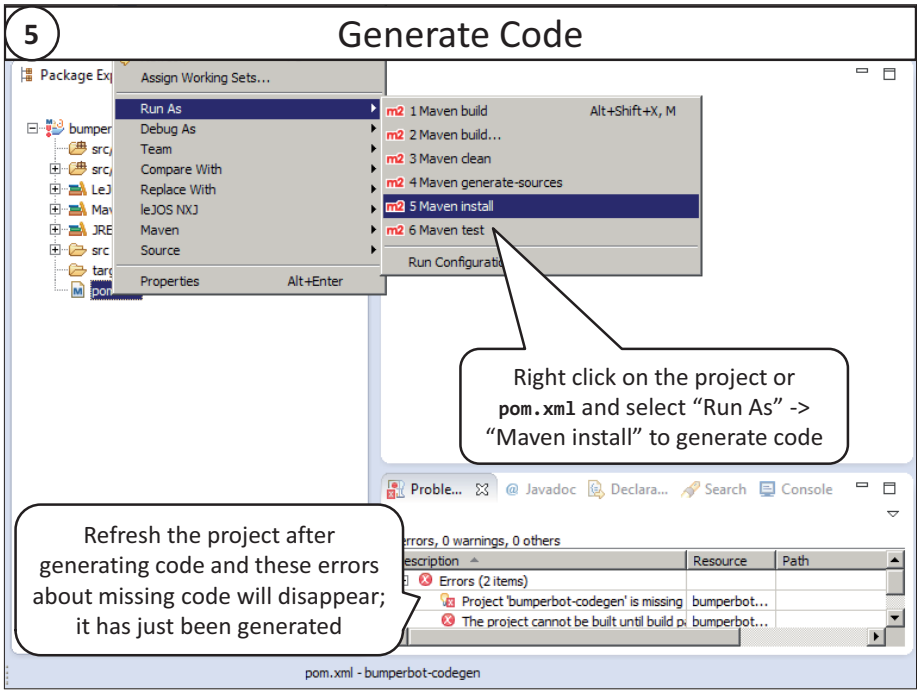
3 Bumper Bot Project Overview

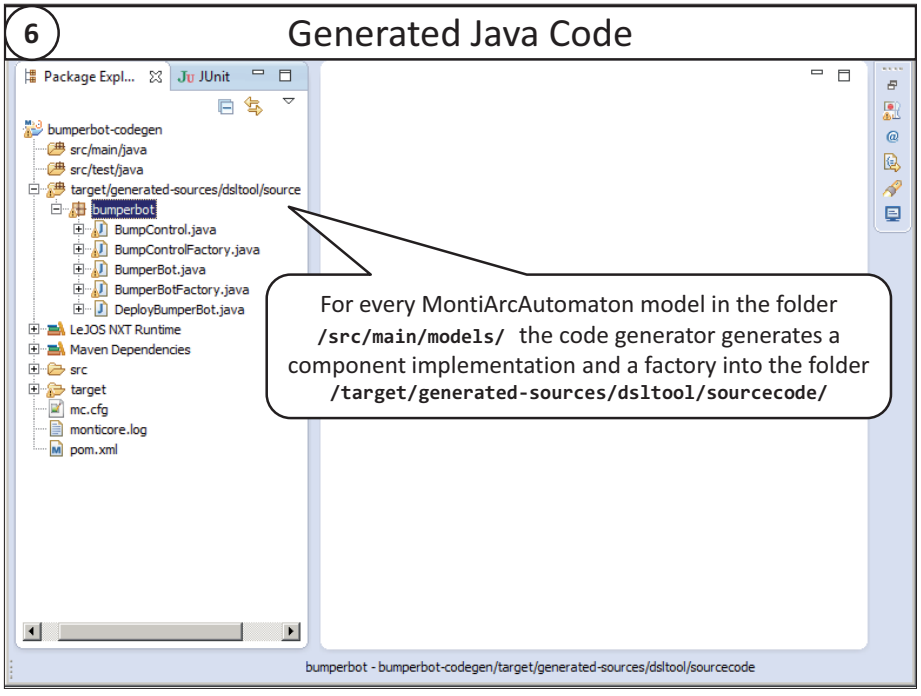


MontiArcAutomaton Java Code Generation

- Invoking the code generator

The code generator is invoked for all MontiArcAutomaton and Class Diagram models placed in **/src/main/models/**.


5
Generate Code
Assign Working Sets...
Run As
Debug As
Team
Compare With
Replace With
leJOS NXJ
Maven
Source
Properties
Alt+Enter
1 Maven build
Alt+Shift+X, M
2 Maven build...
3 Maven clean
4 Maven generate-sources
5 Maven install
6 Maven test
Right click on the project or pom.xml and select "Run As" -> "Maven install" to generate code
Refresh the project after generating code and these errors about missing code will disappear; it has just been generated
Errors (2 items)
Project 'bumperbot-codegen' is missing
The project cannot be built until build p
Resource
Path
pom.xml - bumperbot-codegen


6
Generated Java Code
Package Expl...
JUnit
bumperbot-codegen
src/main/java
src/test/java
target/generated-sources/dsltool/source
bumperbot
BumpControl.java
BumpControlFactory.java
BumperBot.java
BumperBotFactory.java
DeployBumperBot.java
LeJOS NXT Runtime
Maven Dependencies
src
target
mc.cfg
monticore.log
pom.xml
For every MontiArcAutomaton model in the folder /src/main/models/ the code generator generates a component implementation and a factory into the folder /target/generated-sources/dsltool/sourcecode/
bumperbot - bumperbot-codegen/target/generated-sources/dsltool/sourcecode

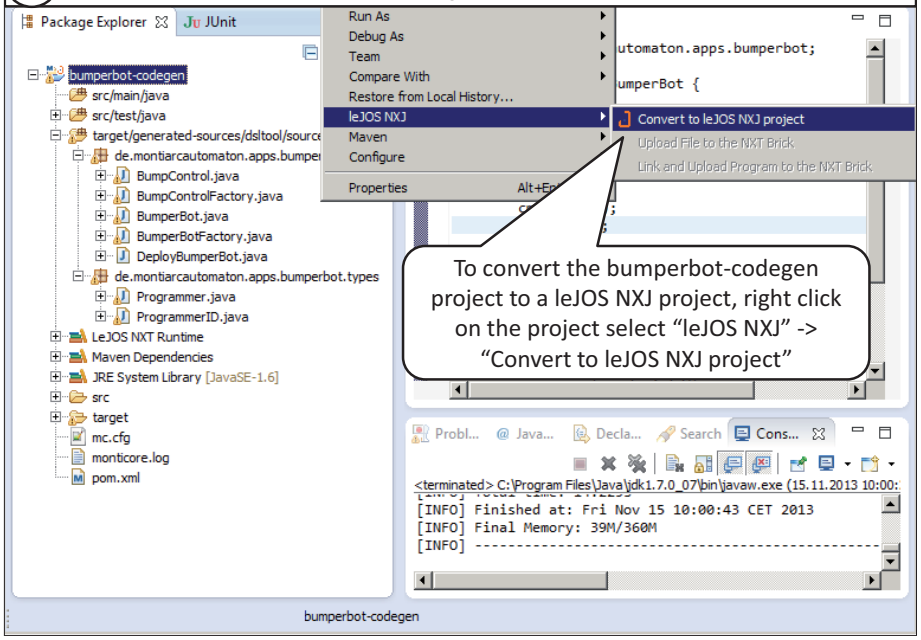
7

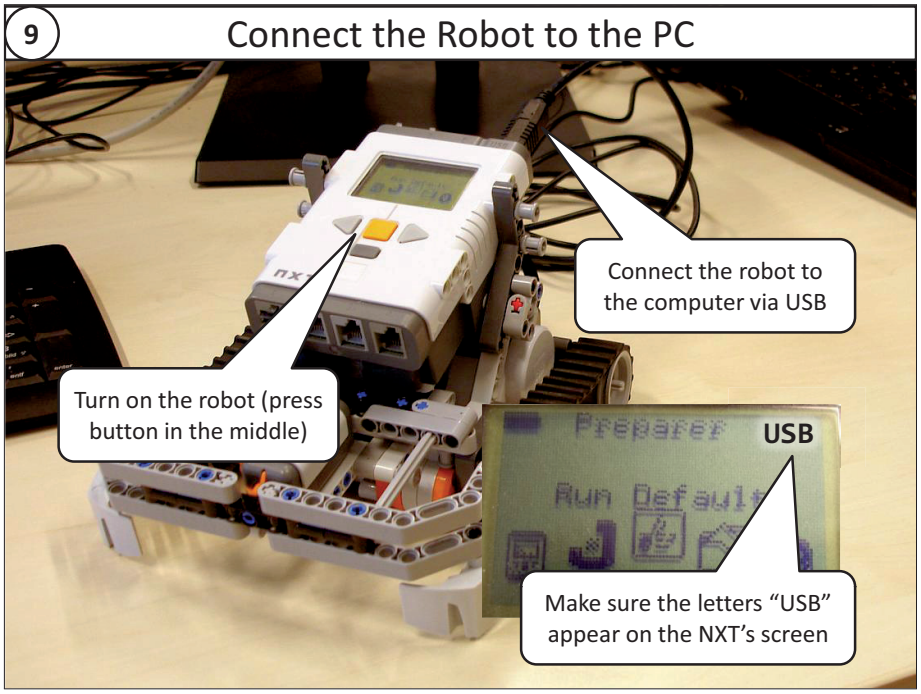
Deploying the Generated Code

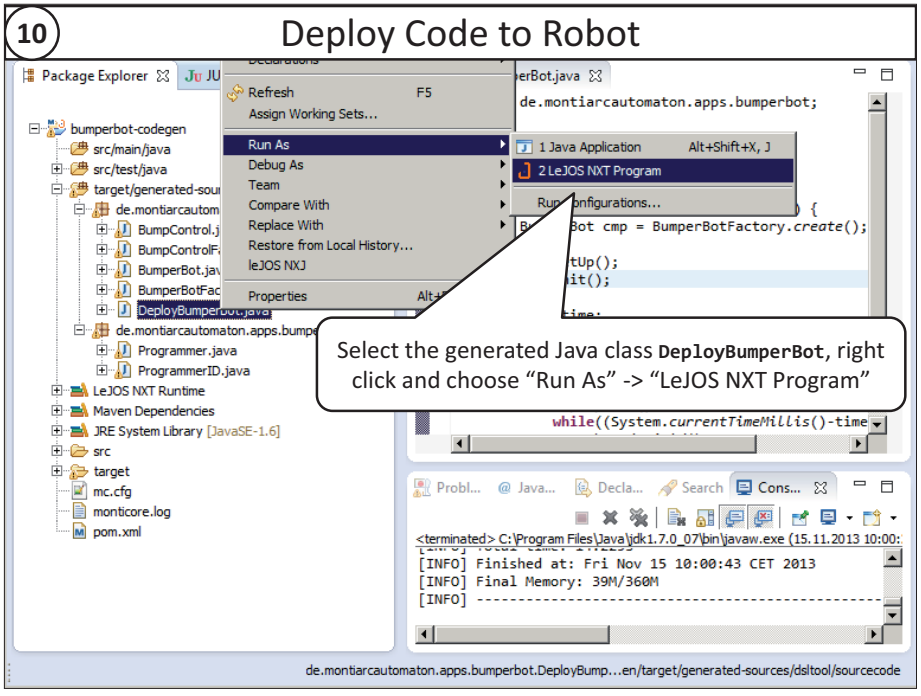
- Convert the project to a leJOS project
- Connect the robot via USB
- Upload the code

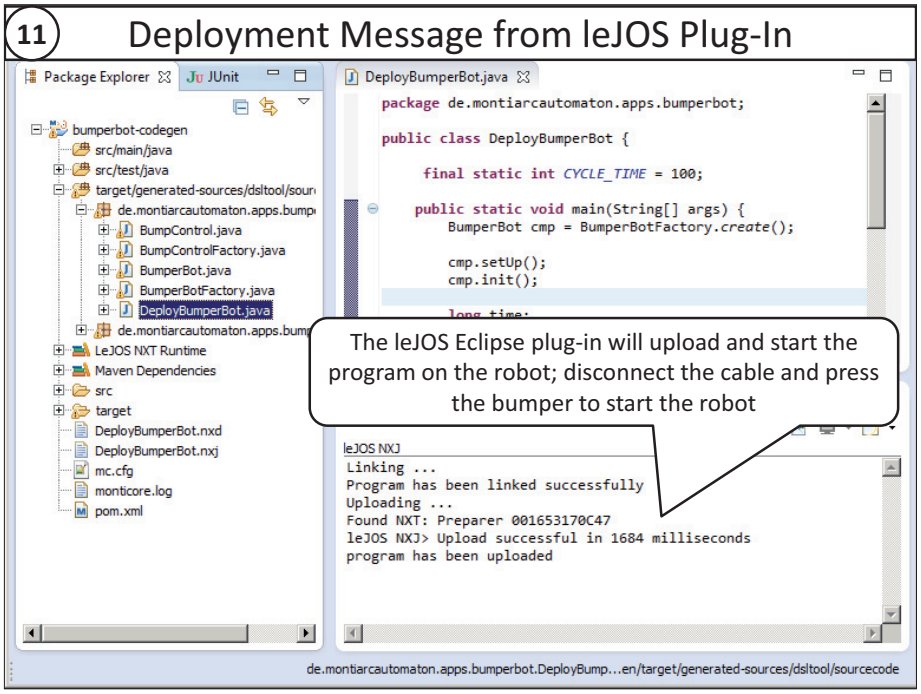
8

Convert Project for leJOS




9
Connect the Robot to the PC
Connect the robot to the computer via USB
Turn on the robot (press button in the middle)
USB
Make sure the letters “USB” appear on the NXT’s screen


10
Deploy Code to Robot
Package Explorer
Refresh F5
Assign Working Sets...
Run As
Debug As
Team
Compare With
Replace With
Restore from Local History...
leJOS NXJ
Properties
1 Java Application Alt+Shift+X, J
2 LeJOS NXT Program
de.montiarcautomaton.apps.bumperbot;
Select the generated Java class DeployBumperBot, right click and choose “Run As” -> “LeJOS NXT Program”
[INFO] Finished at: Fri Nov 15 10:00:43 CET 2013
[INFO] Final Memory: 39M/360M
[INFO] ----------
de.montiarcautomaton.apps.bumperbot.DeployBump...en/target/generated-sources/dsltool/sourcecode


11
Deployment Message from leJOS Plug-In
Package Explorer
JUnit
bumperbot-codegen
src/main/java
src/test/java
target/generated-sources/dsltool/sour
de.montiarcautomaton.apps.bumpe
BumpControl.java
BumpControlFactory.java
BumperBot.java
BumperBotFactory.java
DeployBumperBot.java
de.montiarcautomaton.apps.bump
LeJOS NXT Runtime
Maven Dependencies
src
target
DeployBumperBot.nxd
DeployBumperBot.nxj
mc.cfg
monticore.log
pom.xml
DeployBumperBot.java
package de.montiarcautomaton.apps.bumperbot;
public class DeployBumperBot {
final static int CYCLE_TIME = 100;
public static void main(String[] args) {
BumperBot cmp = BumperBotFactory.create();
cmp.setUp();
cmp.init();
The leJOS Eclipse plug-in will upload and start the program on the robot; disconnect the cable and press the bumper to start the robot
leJOS NXJ
Linking ...
Program has been linked successfully
Uploading ...
Found NXT: Preparer 001653170C47
leJOS NXJ> Upload successful in 1684 milliseconds
program has been uploaded
de.montiarcautomaton.apps.bumperbot.DeployBump...en/target/generated-sources/dsltool/sourcecode

Appendix G.

Complete PumpStation Component and Connector Model

Chapter contains the complete C&C model `PumpStation` used in the example introduced in Section 3.1. The example is published with the AutoFOCUS IDE [BHS99, HF07, wwwe] as part of the AutoFOCUS picture book [wwwc].

The top component of the C&C model is the component `PumpStation`. Each figure displays the parent component with its immediate subcomponents and the names and types of all ports.

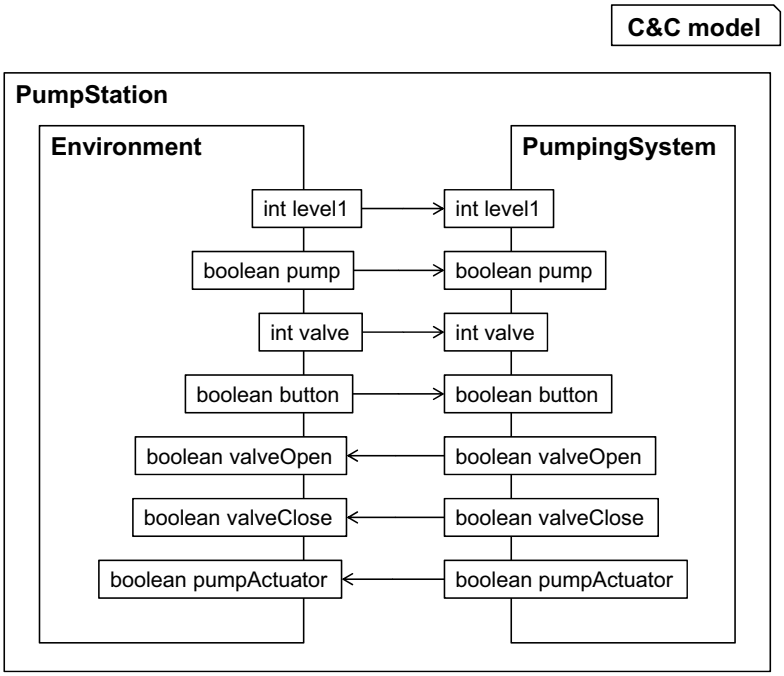


Figure G.1.: The component `PumpStation` with all immediate subcomponents and ports. The component `Environment` is shown in Figure G.2. The component `PumpingSystem` is shown in Figure G.3.

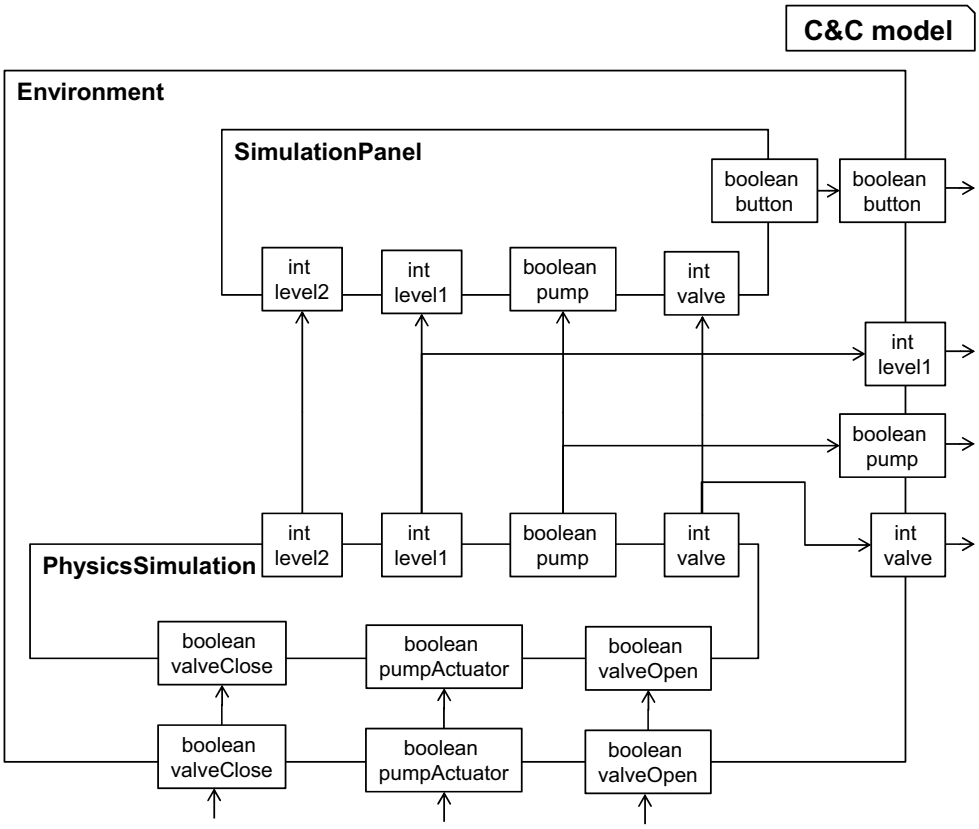


Figure G.2.: The component `Environment` with all immediate subcomponents and ports. The components `SimulationPanel` and `PhysicsSimulation` are atomic.

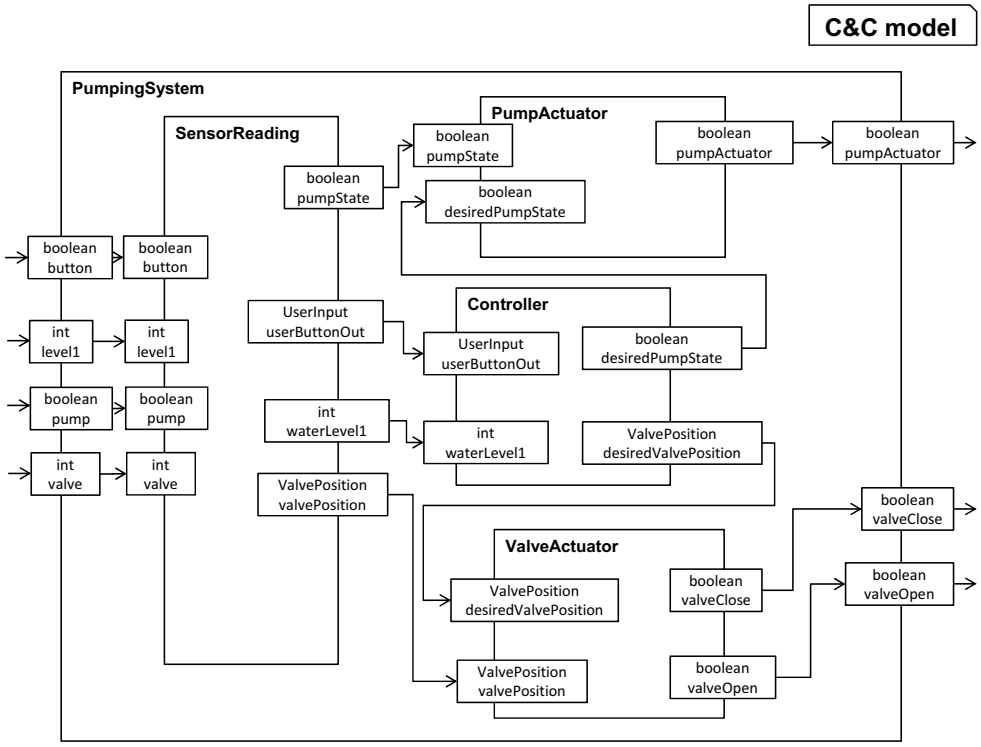


Figure G.3.: The component `PumpingSystem` with all immediate subcomponents and ports. The component `SensorReading` is shown in Figure G.4. The component `Controller` is shown in Figure G.5. The components `PumpActuator` and `ValveActuator` are atomic.

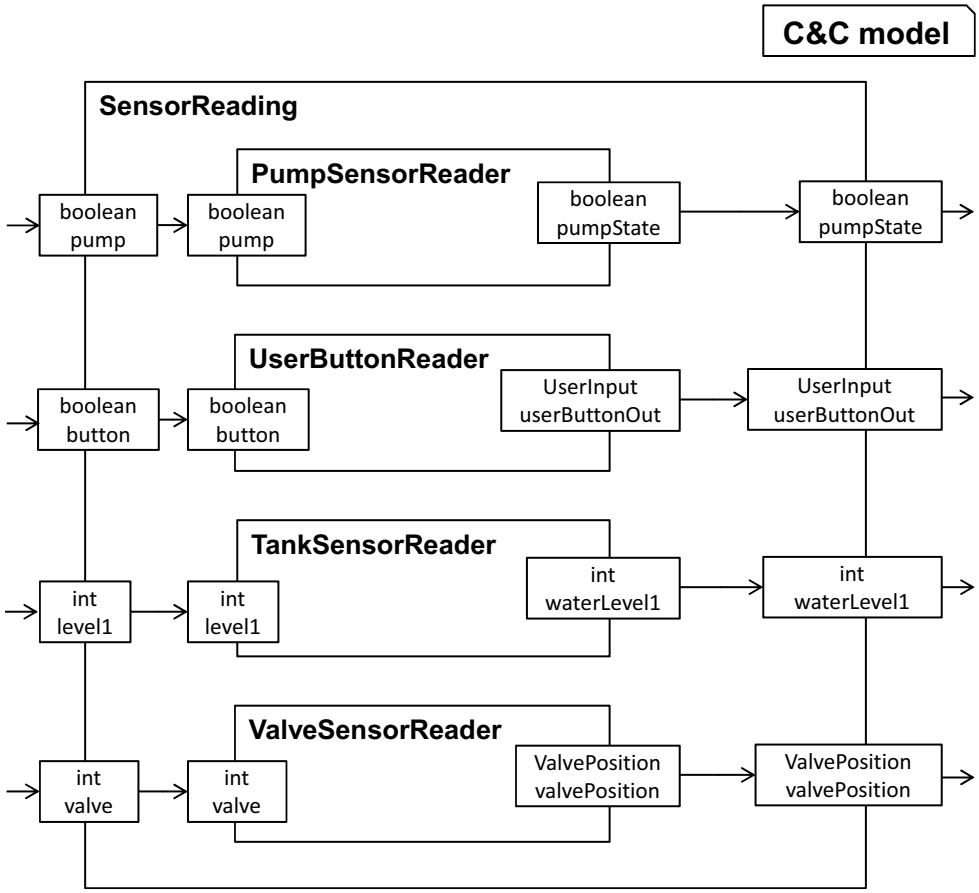


Figure G.4.: The component `SensorReading` with all immediate subcomponents and ports. All subcomponents are atomic.

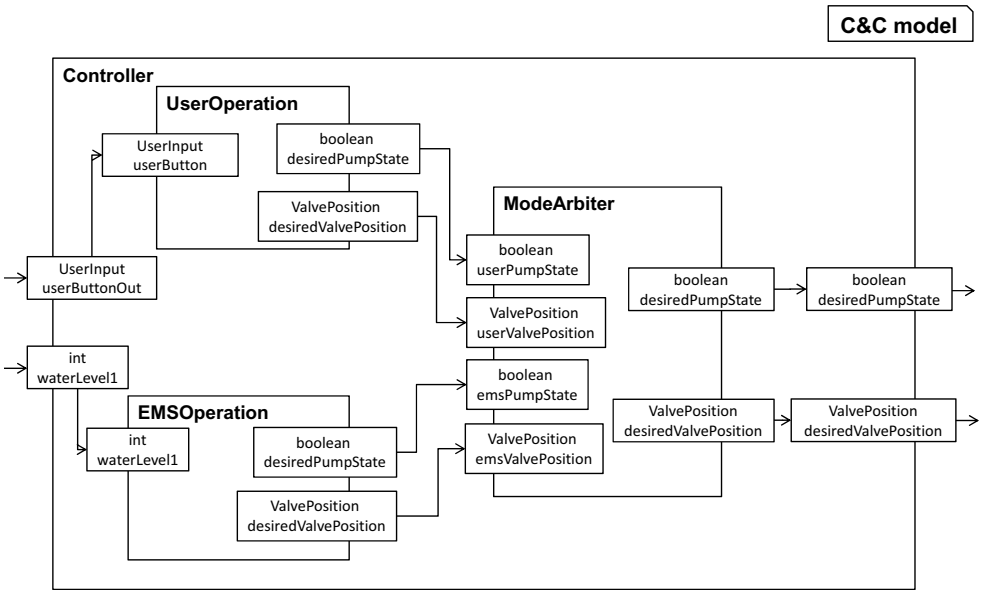


Figure G.5.: The component `Controller` with all immediate subcomponents and ports. All subcomponents are atomic.

Appendix H.

Survey – Helpfulness of Generated Witnesses

This appendix contains the materials used for the survey described in Section 4.4.3.

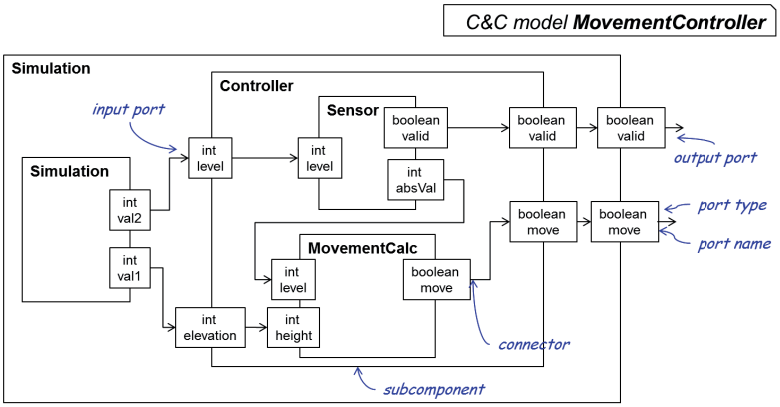
H.1. Reference Materials

The subjects of the survey were presented the C&C model of the pump station as shown in Appendix G and the following two pages of materials that introduce C&C models, C&C views, C&C views satisfaction, and reasons for non-satisfaction.

Component & Connector models

Component and Connector (C&C) models, which are used in many application domains of software engineering, consist of components, ports and connectors, where

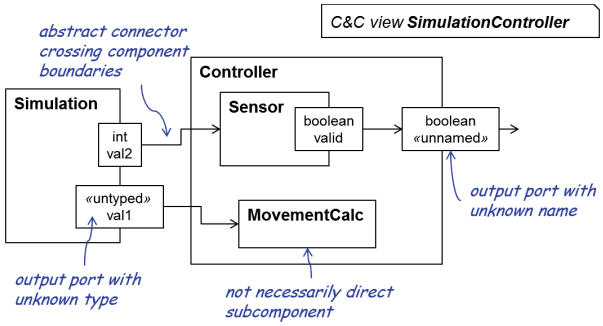
- **components** are named, may have ports and immediate **subcomponents** (components without subcomponents are called **atomic**),
- **ports** are directed (input or output), have a name and a type
- **connectors** are directed and connect two ports of components.



Component & Connector views

C&C views can be used to specify structural properties, e.g., which component should be inside which component, which component should be connected to which component, in an expressive and intuitive way. The elements of C&C views are components, ports and abstract connectors, where:

- **components** are named, may have ports and **subcomponents**,
- **ports** are directed (input or output), have an optional name and an optional type
- **abstract connectors** are directed and connect two ports of components or components directly.



C&C views satisfaction

A C&C model satisfies a C&C view if and only if:

- all **components** shown in the view **exist** in the C&C model,
- component **hierarchy** shown in the view is **preserved** in the C&C model (not necessarily immediate containment),
- all ports in the view have **corresponding ports** (name, type, and direction) in the model, and
- all **abstract connectors** in the view have corresponding chains of connectors in the model

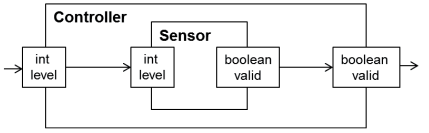
Reasons for non-satisfaction

Directly derived reasons for non-satisfaction:

- derived → **Missing Component:** the view contains a component that does not exist in the C&C model
- derived → **Hierarchy Mismatch**: the view contains two components that in the C&C model are in a different containment relation (reverse containment, not contained, contained)
- derived → **Interface Mismatch:** the view contains a component with a port that does not exist in the C&C model (no full match of name, type, and direction)
- derived → **Missing Connection:** the view contains an abstract connector that has no corresponding concrete chain of connectors in the C&C model.

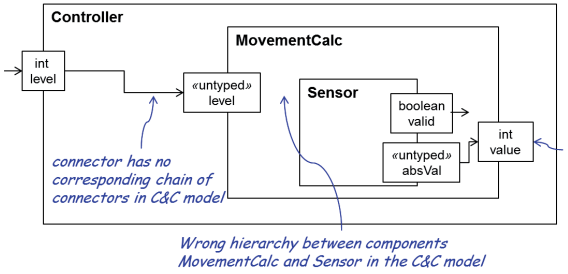
Chains of connectors are sequences of connectors where the target port of the previous connector is the source port of the next connector. Chains end when they lead to a component that does not forward the data via internal connectors.

*C&C view **SensorOnly***



The C&C model MovementCon-troller from the previous page **satisfies** the C&C view SensorOnly.

*C&C view **SensorAndCalc***



The C&C model MovementCon-troller from the previous page **does not satisfy** the C&C view SensorAndCalc.

H.2. Printed Survey

The following pages contain a printed version of the online survey presented to the subjects. The links on the pages point to a PDF displaying the C&C model of the pump station shown in Appendix G and to a PDF with the document presented in Section H.1.

ArcVCheck

There are 10 questions in this survey

Intro

[Intro]

Component and connector (C&C) models are used in many application domains of software engineering.

C&C views can be used to specify structural properties of C&C models, e.g., which component should be inside which component, which component should be connected to which component, in an expressive and intuitive way.

An introduction to C&C models and C&C views is given in this PDF.

All the tasks in the exercise are based on the same C&C model of a pump station (download the PDF here).

Are you ready to start the exercise?

*

Please choose **only one** of the following:

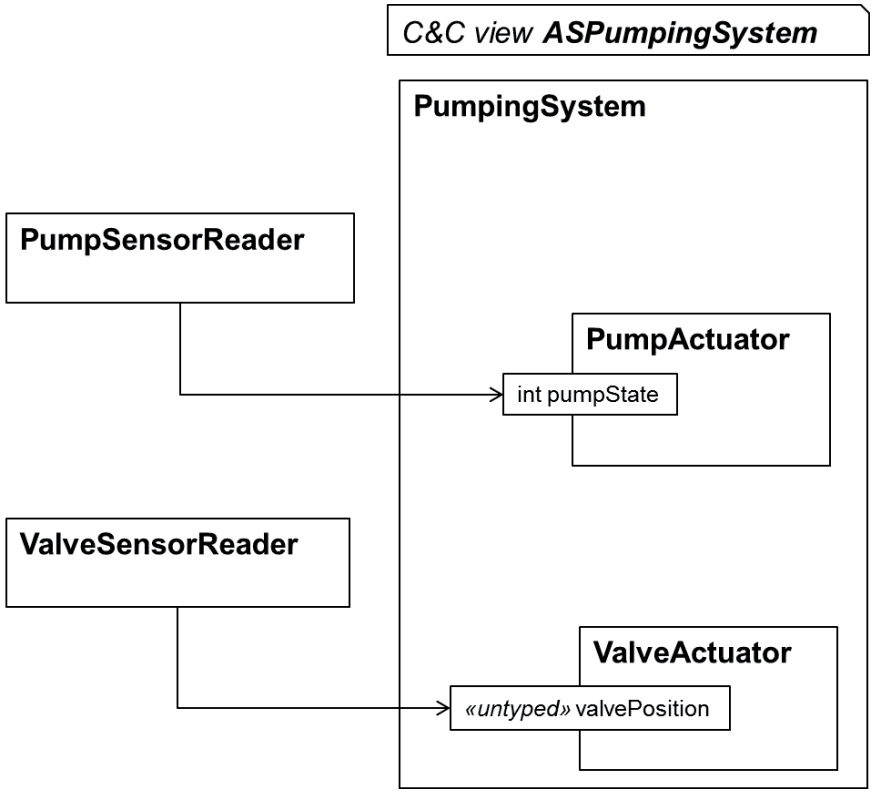
○ Yes, I am ready.

C&C Views Verification

In this part we will ask you to check whether a given C&C model satisfies three C&C views.

[SAT1]

Does the C&C model of the pump station (same as before) satisfy this view?



Check either "Yes, ..." or check "No, ..." for ALL reasons for non-satisfaction you find.

*

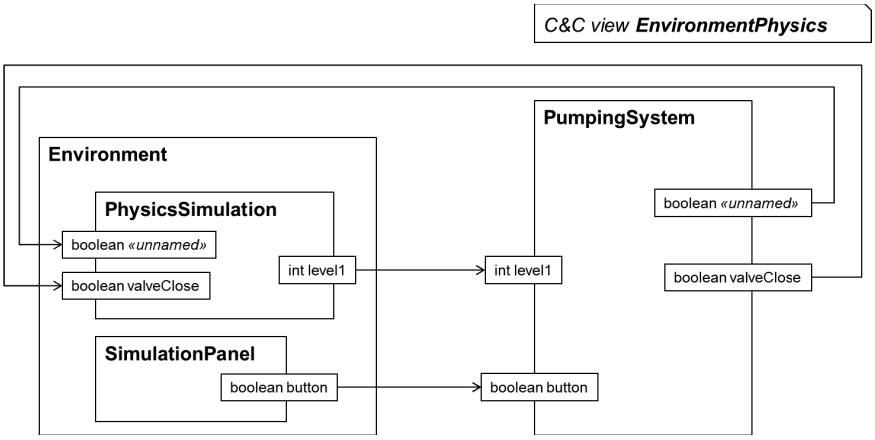
Bitte wählen Sie zwischen 1 und 4 Antworten aus.

Please choose **all** that apply:

- ☐ Yes, the C&C model satisfies the view.
- ☐ No, there is a Hierarchy Mismatch
- ☐ No, there is a Missing Component
- ☐ No, there is a Interface Mismatch
- ☐ No, there is a Missing Connector

[SAT2]

Does the C&C model of the pump station (same as before) satisfy this view?



Check either "Yes, ..." or check "No, ..." for ALL reasons for non-satisfaction you find.

*

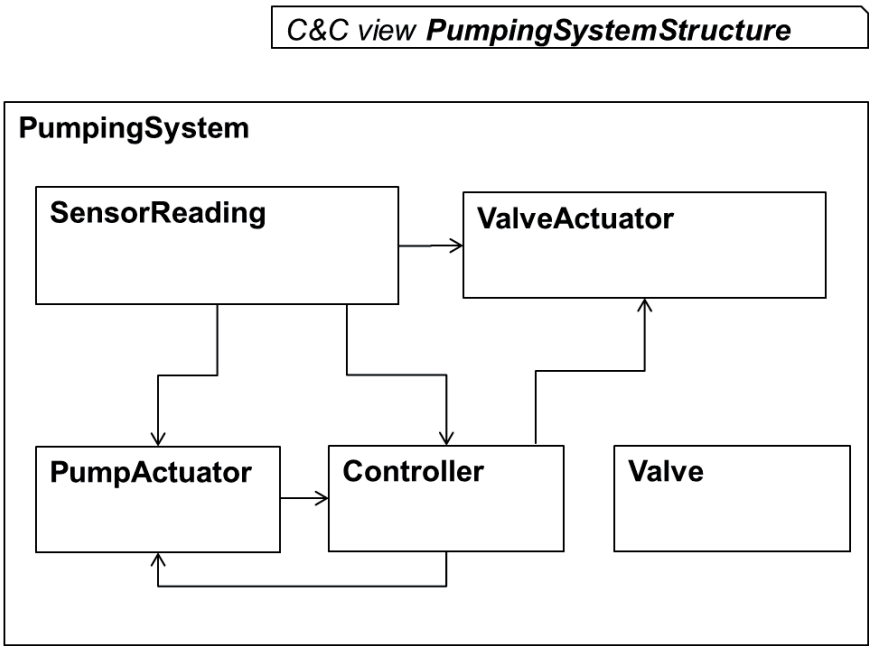
Bitte wählen Sie zwischen 1 und 4 Antworten aus.

Please choose **all** that apply:

- ☐ Yes, the C&C model satisfies the view.
- ☐ No, there is a Hierarchy Mismatch
- ☐ No, there is a Missing Component
- ☐ No, there is a Interface Mismatch
- ☐ No, there is a Missing Connector

[SAT3]

Does the C&C model of the pump station (same as before) satisfy this view?



Check either "Yes, ..." or check "No, ..." for ALL reasons for non-satisfaction you find.

*

Bitte wählen Sie zwischen 1 und 4 Antworten aus.

Please choose **all** that apply:

- ☐ Yes, the C&C model satisfies the view.
- ☐ No, there is a Hierarchy Mismatch
- ☐ No, there is a Missing Component
- ☐ No, there is a Interface Mismatch
- ☐ No, there is a Missing Connector

[Correctness]

Please rate your confidence that you correctly identified whether the C&C views were satisfied by the C&C model. Rate your confidence regardless of the reasons you identified.

*

Please choose **only one** of the following:

- ○ I am 100% right
- ○ I might have decided wrong on 1 of 3
- ○ I might have decided wrong on 2 of 3
- ○ I don't think I decided satisfaction right for any of the views

[Completeness]

Please rate your confidence in finding all reasons for non-satisfaction.

*

Please choose **only one** of the following:

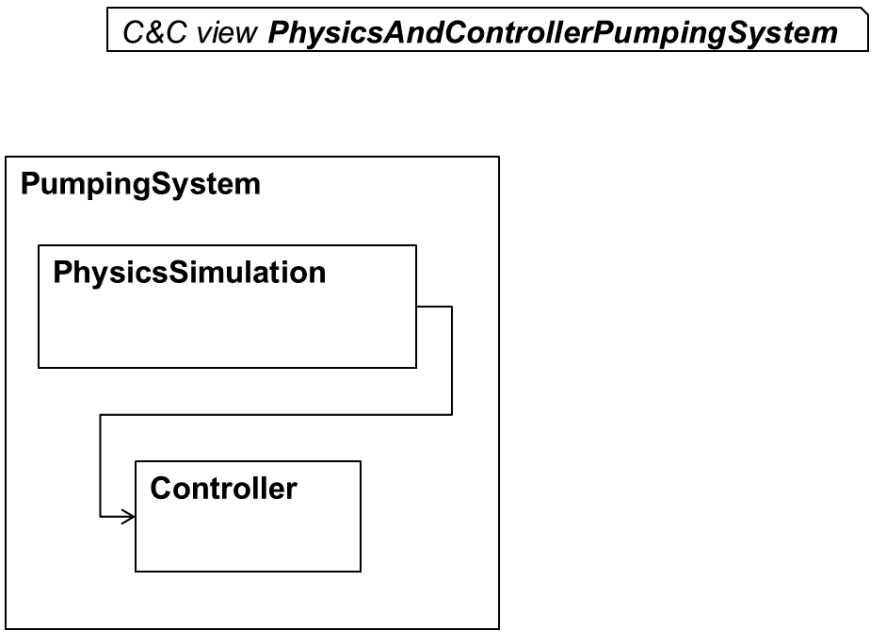
- ○ I always found all reasons
- ○ I might have missed one
- ○ I probably missed most of them

Helpful Witnesses

In this second part of the exercise you will see a view and witnesses that support the verification result. We will ask you whether these are helpful to you.

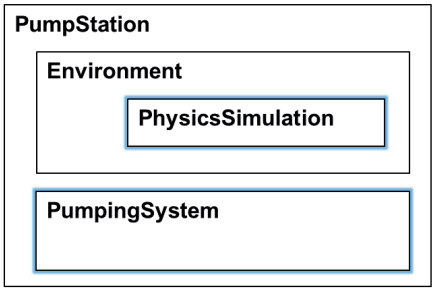
[Witness1]

Consider the C&C model of the pump station (same as before) and the C&C view below:



The C&C model does not satisfy this C&C view. Here are two witnesses that demonstrate separate reasons for non-satisfaction:

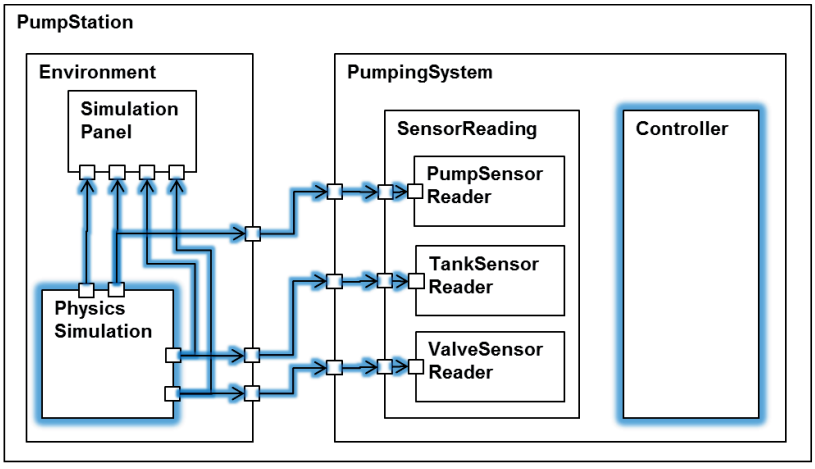
First witness for non-satisfaction:



Hierarchy Mismatch: *the witness shows the containment relation in the C&C model.*

Components PumpingSystem and PhysicsSimulation are independent in the C&C model PumpStation but not independent in view PCPumpingSystem.

Second witness for non-satisfaction:



Missing Connection: *the witness shows all outgoing connectors from component PhysicsSimulation.*

The C&C model PumpStation is missing connection from component Physics-Simulation to component Controller (from unnamed port to unnamed port).

Do you find these witnesses helpful in understanding the verification result (non-satisfaction)?

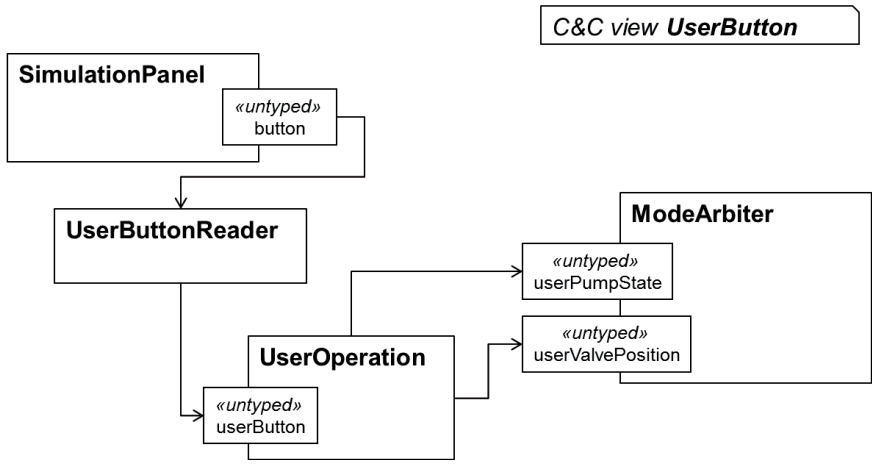
*

Please choose **only one** of the following:

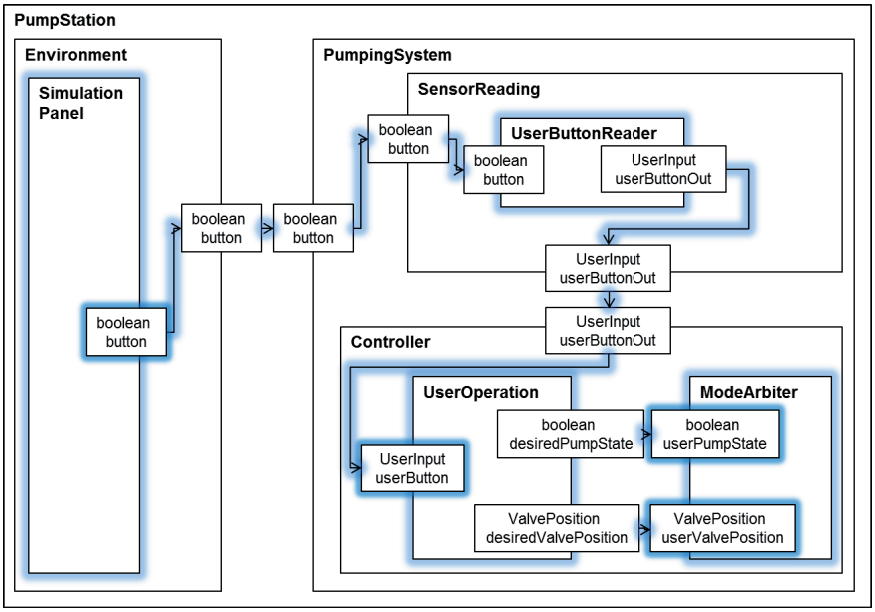
- ○ Yes, the witnesses are very helpful
- ○ Yes, they help me
- ○ I don't know
- ○ No, they are not helpful
- ○ No, the witnesses are misleading

[Witness2]

Consider the C&C model of the pump station (same as before) and the C&C view below:



The C&C model satisfies this C&C view. Here is a witnesses for satisfaction:



Satisfaction: *The witness shows all elements from the view in the C&C model.*

Do you find this witnesses helpful in understanding the verification result (satisfaction)?

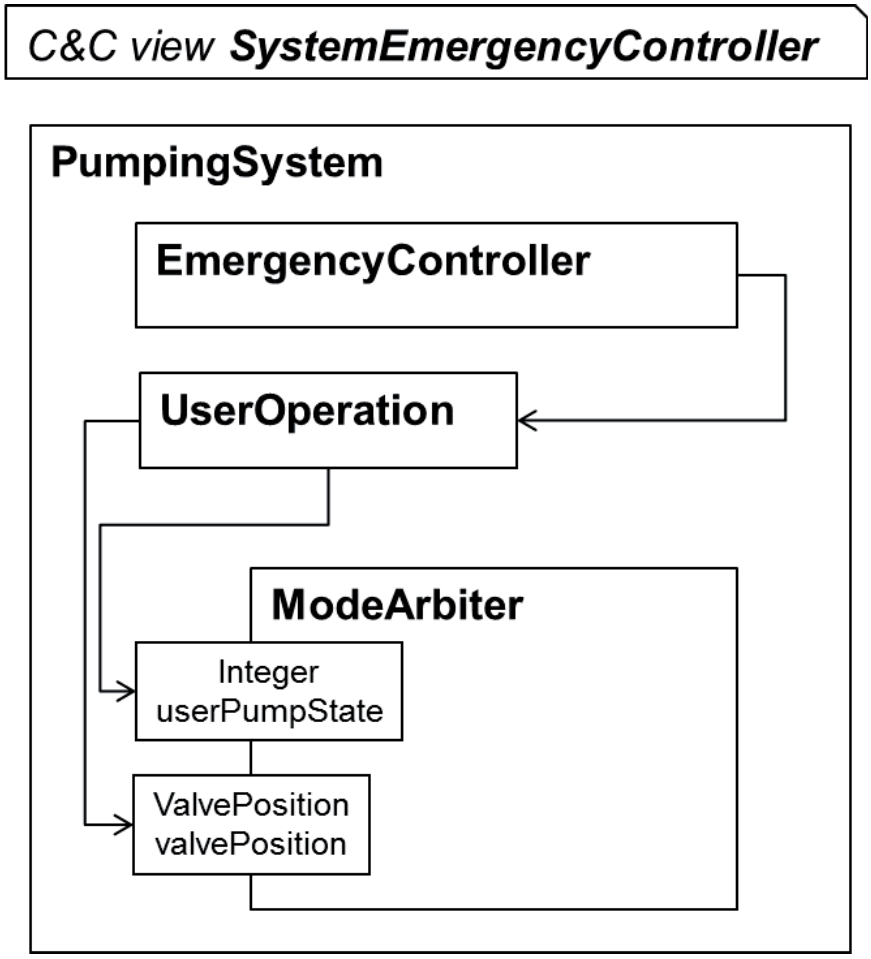
*

Please choose **only one** of the following:

- ○ Yes, the witnesses are very helpful
- ○ Yes, they help me
- ○ I don't know
- ○ No, they are not helpful
- ○ No, the witnesses are misleading

[Witness3]

Consider the C&C model of the pump station (same as before) and the C&C view below:

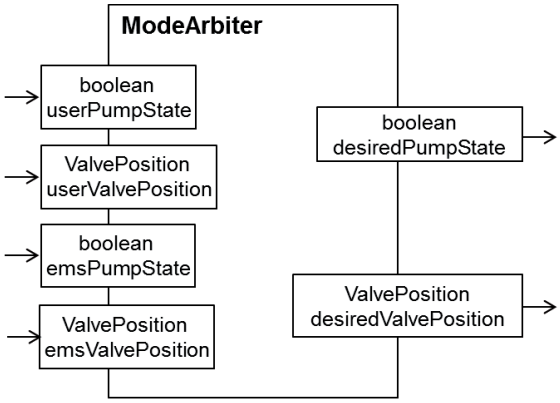


The C&C model does not satisfy this C&C view. Here are four witnesses that demonstrate separate reasons for non-satisfaction.

First witness for non-satisfaction:

Missing Component: Component EmergencyController of view SystemEmergencyController is missing in C&C model PumpStation.

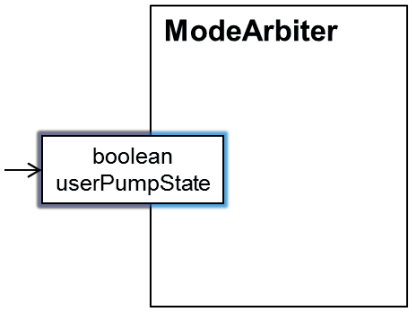
Second witness for non-satisfaction:



Interface Mismatch: *the witness shows all ports of component ModeArbiter.*

No match for port valvePosition of component ModeArbiter.

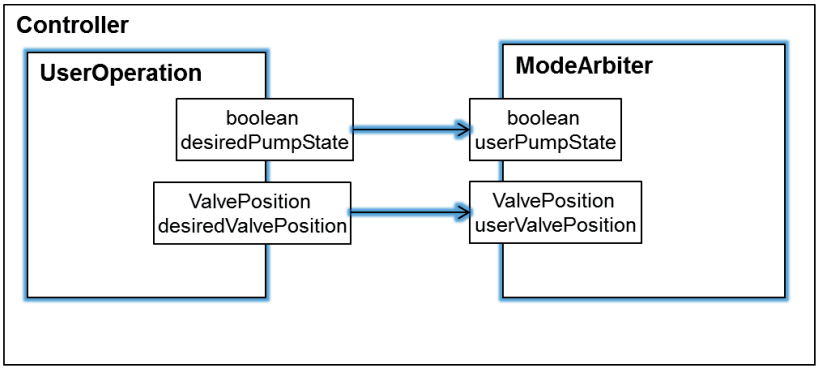
Third witness for non-satisfaction:



Interface Mismatch: *the witness shows the port and its correct type.*

Wrong type for port userPumpState of component ModeArbiter (Integer).

Fourth witness for non-satisfaction:



Missing Connection: *the witness shows all outgoing connectors from component UserOperation.*

C&C model PumpStation is missing connection from component UserOperation to component ModeArbiter (from unnamed port to port valvePosition)

Do you find these witnesses helpful in understanding the verification result (non-satisfaction)?

*

Please choose **only one** of the following:

- ○ Yes, the witnesses are very helpful
- ○ Yes, they help me
- ○ I don't know
- ○ No, they are not helpful
- ○ No, the witnesses are misleading

Remarks

[Remarks]

Please note any comments you may have about C&C models, C&C views, and C&C witnesses.

Please write your answer here:

Appendix I.

Complete C&C Views Synthesis Alloy Translation Example

This appendix contains the complete translation result of the C&C views synthesis example presented in Section 5.1.1. The views used as input for the translation are shown in Figure 5.1, Figure 5.2, and Figure 5.3. The specification used is specification S_1 from Section 5.1.1 with a port scope of 18, no library components, and no selected architectural style.

The example illustrates the translation rules introduced in Section 5.3.4.

Alloy

```
// Alloy Module for view synthesis of views:
// ASDependence, BodySensorIn, BodySensorOut, RJFunction,
    SensorConnections, RJStructure

module viewSynthesis

////////////////////////////////////////////////
// Part I: C&C models
////////////////////////////////////////////////

abstract sig Direction {}
one sig IN extends Direction{}
one sig OUT extends Direction{}

sig PortName {}

sig Type {}

abstract sig Component {
  ports : set Port,
  subComponents : set Component,

  //derived
  parent : lone Component
}

fact subComponentsAndParents {
```

```
27    all ch, par : Component|
28      (ch in par.subComponents iff ch.parent = par)
29  }
30
31  fact subComponentsAcyclic {
32    no comp : Component |
33      comp in comp.^subComponents
34  }
35
36  sig Port {
37    type : one Type,
38    name: one PortName,
39    direction: one Direction,
40
41    receivingPorts : set Port,
42
43    //derived
44    owner : one Component,
45    sendingPort : lone Port
46  } {
47    this != sendingPort
48    this not in receivingPorts
49  }
50
51  fact portsAndOwners {
52    all cmp : Component | all port: cmp.ports |
53      cmp = port.owner
54  }
55
56  fact portsAndSender {
57    all disj sender, receiver :Port |
58      //(sender = receiver) or
59      (sender = receiver.sendingPort iff receiver in sender.
           receivingPorts)
60  }
61
62  fact portsOfComponentHaveUniqueNames {
63    all c:Component | all disj p1, p2 : c.ports |
64      (p1.name != p2.name)
65  }
66
67  fact portsConnectedLegally {
68    all sender: Port | all receiver : (sender.receivingPorts-sender)
         |
69      (receiver.type = sender.type and
70        // direct connector
71        ( (receiver.owner = sender.owner and sender.direction = IN
             and receiver.direction= OUT) or
72        // toChildConnector
```

```
73        (receiver.owner in sender.owner.subComponents and sender.
              direction = IN and receiver.direction= IN) or
74        // fromChildConnector
75        (sender.owner in receiver.owner.subComponents and sender.
              direction = OUT and receiver.direction= OUT) or
76        // subComponentConnector
77        (receiver.owner != sender.owner and sender.owner.parent =
              receiver.owner.parent and sender.direction = OUT and
              receiver.direction= IN) ) )
78  }
79
80  // instance does not contain any unused atoms
81  fact allPortsOwned {
82    Port in Component.ports
83  }
84
85
86  ////////////////////////////////////////////////
87  // Part II: Views
88  ////////////////////////////////////////////////
89
90  // Containment and Independence
91  ////////////////////////////////////////////////
92
93  pred contains [parent: Component, child: Component] {
94    child in parent.^(subComponents)
95  }
96
97  // components are not contained in each other when shown in the
        same view side by side
98  pred independent[p: Component, child: Component] {
99    not (contains[p, child] or contains[child, p])
100 }
101
102 pred independentSet [components : set Component] {
103   all disj c1, c2 : components |
104     ( (no c1.subComponents)  or
105         (not contains[c1, c2]) )
106 }
107
108 pred atomicComponent[cmp: Component] {
109   // no children
110   no cmp.subComponents
111   // no connectors starting in incoming ports
112  no (cmp.ports & direction.IN).receivingPorts
113 }
114
115 // Connectedness
116 ////////////////////////////////////////////////
```

```
117
118 pred connected[sender: Component, receiver: Component] {
119   some p : receiver.ports |
120     p in sender.ports. ^receivingPorts
121 }
122
123 pred connectedWithPortNames[sender: Component, sendName : PortName,
        receiver: Component, recvName: PortName] {
124   some sp : sender.ports | some rp : receiver.ports |
125     rp.name = recvName and
126     rp in sp.^receivingPorts and
127     sp.name = sendName and
128     sp in rp.^~receivingPorts
129 }
130
131 pred connectedWithSenderPortName[sender: Component, sendName :
      PortName, receiver: Component] {
132   some recvName : receiver.ports.name |
133     connectedWithPortNames[sender, sendName, receiver, recvName]
134 }
135
136 pred connectedWithReceiverPortName[sender: Component, receiver:
      Component, recvName : PortName] {
137   some sendName : sender.ports.name |
138     connectedWithPortNames[sender, sendName, receiver, recvName]
139 }
140
141 // Ports of Components
142 ////////////////////////////////////////////////
143
144 pred untypedPort[cmp : Component, dir: Direction, portName :
      PortName] {
145   some port : cmp.ports |
146     port.direction = dir and
147     port.name = portName
148 }
149
150 pred unnamedPort[cmp : Component, dir: Direction, pType : Type] {
151   some port : cmp.ports |
152     port.direction = dir and
153     port.type = pType
154 }
155
156 pred portOfComponent[cmp : Component, dir: Direction, pType: Type,
      portName : PortName] {
157   some port : cmp.ports |
158     port.direction = dir and
159     port.name = portName and
160     port.type = pType
```

```
161 }
162
163 pred interfaceComplete[cmp:Component, portNames : set PortName] {
164   cmp.ports.name = portNames
165 }
166
167 fact oneRoot{
168   one (Component - Component.subComponents)
169 }
170
171
172 ////////////////////////////////////////////////////////////
173 // Signatures for all components, port names and types
174 ////////////////////////////////////////////////////////////
175
176 // Concrete components from all views
177 lone sig RotationalJoint extends Component {}
178 lone sig Body extends Component {}
179 lone sig Actuator extends Component {}
180 lone sig Sensor extends Component {}
181 lone sig ServoValve extends Component {}
182 lone sig Cylinder extends Component {}
183 lone sig Joint extends Component {}
184 lone sig JointLimiter extends Component {}
185
186 // Port names used in all views
187 one sig val1 extends PortName {}
188 one sig f1 extends PortName {}
189 one sig val2 extends PortName {}
190 one sig f2 extends PortName {}
191 one sig angle extends PortName {}
192
193 // Types used in all views
194 one sig my_int extends Type {}
195 one sig my_float extends Type {}
196
197
198 ////////////////////////////////////////////////
199 // Predicates for the views
200 ////////////////////////////////////////////////
201
202 pred ASDependence {
203   (some Component)
204   and one Body
205   and one Actuator
206   and one Sensor
207
208   // all independent sets
209   and independentSet[Actuator + Sensor]
```

```
210
211   // containment relation
212   and contains[Body, Actuator + Sensor]
213
214   // ports of components
215
216   // connections between components
217 }
218
219 pred BodySensorIn {
220   (some Component)
221   and one Body
222   and one Actuator
223   and one Sensor
224   and one JointLimiter
225   and one Joint
226
227   // all independent sets
228   and independentSet[Actuator + Joint + JointLimiter + Sensor]
229
230   // containment relation
231   and contains[Body, Actuator + Joint + JointLimiter + Sensor]
232
233   // ports of components
234   // ports of components Actuator
235   and untypedPort[Actuator, IN, f1]
236   and untypedPort[Actuator, IN, f2]
237
238   // connections between components
239   and connectedWithReceiverPortName[Body, Actuator, f1]
240   and connectedWithReceiverPortName[Body, Actuator, f2]
241   and connected[Actuator, Joint]
242   and connected[Sensor, JointLimiter]
243   and connected[JointLimiter, Actuator]
244 }
245
246 pred BodySensorOut {
247   (some Component)
248   and one Body
249   and one Actuator
250   and one Sensor
251   and one JointLimiter
252   and one Joint
253
254   // all independent sets
255   and independentSet[Actuator + Joint + JointLimiter]
256   and independentSet[Body + Sensor]
257
258   // containment relation
```

```
259   and contains[Body, Actuator + Joint + JointLimiter]
260
261   // ports of components
262   // ports of components Actuator
263   and untypedPort[Actuator, IN, f1]
264   and untypedPort[Actuator, IN, f2]
265
266   // connections between components
267   and connectedWithReceiverPortName[Body, Actuator, f1]
268   and connectedWithReceiverPortName[Body, Actuator, f2]
269   and connected[Actuator, Joint]
270   and connected[Sensor, JointLimiter]
271   and connected[JointLimiter, Actuator]
272 }
273
274 pred RJFunction {
275   (some Component)
276   and one RotationalJoint
277   and one Actuator
278   and one Sensor
279   and one Cylinder
280
281   // all independent sets
282   and independentSet[Sensor + Actuator + Cylinder]
283
284   // containment relation
285   and contains[RotationalJoint, Sensor + Actuator + Cylinder]
286
287   // ports of components
288
289   // connections between components
290   and connected[Sensor, Actuator]
291 }
292
293 pred SensorConnections {
294   (some Component)
295   and one Sensor
296   and one JointLimiter
297   and one Cylinder
298
299   // all independent sets
300   and independentSet[Sensor + Cylinder + JointLimiter]
301
302   // containment relation
303
304   // ports of components
305   // ports of components Sensor
306   and portOfComponent[Sensor, OUT, my_float, val1]
307   and portOfComponent[Sensor, OUT, my_int, val2]
```

```
308
309   // connections between components
310   and connectedWithSenderPortName[Sensor, val1, JointLimiter]
311   and connectedWithSenderPortName[Sensor, val2, Cylinder]
312 }
313
314 pred RJStructure {
315   (some Component)
316   and one RotationalJoint
317   and one Body
318   and one ServoValve
319   and one Cylinder
320
321   // all independent sets
322   and independentSet[ServoValve + Body + Cylinder]
323
324   // containment relation
325   and contains[RotationalJoint, ServoValve + Body + Cylinder]
326
327   // ports of components
328   // ports of components Cylinder
329   and portOfComponent[Cylinder, IN, my_float, angle]
330
331   // connections between components
332   and connected[RotationalJoint, ServoValve]
333   and connected[ServoValve, Body]
334   and connectedWithReceiverPortName[Body, Cylinder, angle]
335   and connected[Cylinder, Body]
336 }
337
338 fact libraryComponents {
339 }
340
341 pred specification {
342   ((not ASDependence)) and (BodySensorIn or BodySensorOut) and (
        RJFunction) and (SensorConnections) and (RJStructure)
343 }
344
345 ////////////////////////////////////////////////////
346 // Run command
347 ////////////////////////////////////////////////////
348
349 run specification for 6 but exactly 18 Port, 8 Component
```

Listing I.1: Alloy module translated from specification S_1 presented in Section 5.1.1.

Appendix J.

MontiArcAutomaton Grammar for Human Reading

The modeling language MontiArcAutomaton is an extension of the architecture description language MontiArc for the modeling of cyber-physical systems. The grammar of the MontiArc modeling language is provided in [HRR12, Appendix A.2].

The grammar of the MontiArcAutomaton language is shown in Listing J.1. It has been adapted from the original MontiCore grammar to a more readable form by removing some technical annotations from the grammar. The complete grammar is provided in [RRW14].

Details on the MontiCore grammar format are available in [Kra10, KRV10].

MontiCore Grammar

```
grammar MontiArcAutomaton extends MontiArc {

  VariableDeclaration implements ArcElement =
      "var"? Type Variable ("," Variable)* ";";

  Variable = Name ("=" Value)?;

  Automaton implements ArcElement =
      Stereotype? "automaton" Name? "{"
          (StateDeclaration |
           InitialStateDeclaration |
           Transition)*
       "}" ;

  StateDeclaration = "state" State ( "," State )* ";" ;

  State = Stereotype? Name;

  InitialStateDeclaration =
    "initial" Name ("," Name)* ("/" IOBlock)? ";";

  Transition = source:Name ("->" target:Name)?
                              Guard? input:IOBlock?
                              ("/" output:IOBlock)? ";";

  Guard = "[" (kind:Name ":")?
          guardExpression:InvariantContent(parameter kind) "]";

  IOBlock    = ( "{" IOAssignment ("," IOAssignment)* "}" )
             | (     IOAssignment ("," IOAssignment)* );

  IOAssignment = (Name "=")? IOStream ("|" IOStream)*;

  IOStream = Value (":" Value)*;

  NoData implements Value = "--";
}
```

Listing J.1: The grammar of MontiArcAutomaton extending the MontiArc ADL with automata and local variables inside components.

Appendix K.

MontiArcAutomaton Specification Suite Grammar for Human Reading

The grammar of the modeling language for MontiArcAutomaton specification suites and specification checks is shown in Listing J.1. It has been adapted from the original MontiCore grammar to a more readable form by removing some technical annotations from the grammar.

Details on the MontiCore grammar format are available in [Kra10, KRV10].

MontiCore Grammar

```
1  grammar MAASpecification {
2
3    SpecificationSuite =
4      Stereotype? "suite" Name "{"
5        CTDefCheck+
6      "}";
7
8    CTDefCheck = "check" Name ":"
9      (negate:["not"])?
10     left:Conjunct Relation right:Conjunct";";
11
12   Conjunct = CTDef:Name ("and" CTDef:Name)*;
13
14   Relation = (refinement:["refines"] | equality:["equals"]);
15 }
```

Listing K.1: The grammar of the MontiArcAutomaton specification language to define specification suites containing specification checks.

Appendix L.

Complete MontiArcAutomaton Verification Mona Translation Example

This appendix contains complete input models and their translation results for the MontiArcAutomaton verification implementation based on the translation rules introduced in Section 7.3. Translation results for a composed component are shown in Section L.1. Translation results for an atomic component with an automaton are shown in Section L.2.

L.1. Example Translation of a Composed Component

A complete translation of the composed component `BumperBotESController` shown in Listing L.1 in MontiArcAutomaton syntax is shown in Listing L.2 in Mona syntax. The example has been used in Section 7.3.3 to illustrate the translation rules for composed components.

MontiArcAutomaton

```
1  package bumperbotv2;
2
3  import lib.Timer;
4  import lib.types.*;
5  import bumperbotemergency.BumpControlES;
6
7  component BumperBotESController {
8
9    port
10     in Boolean emgStp,
11     in Boolean bump,
12     out MotorCmd rMot,
13     out MotorCmd lMot;
14
15   component Timer timer;
16
17   component BumpControlES bces;
18
```

```
19   connect emgStp -> bces.emgStp;
20   connect bump -> bces.bump;
21   connect bces.lMot -> lMot;
22   connect bces.rMot -> rMot;
23   connect timer.ts -> bces.ts;
24   connect bces.tc -> timer.tc;
25
26 }
```

Listing L.1: MontiArcAutomaton model of the composed component `BumperBotESController`.

Mona

```
1  # Mona predicate representing component BumperBotESController
2  #DEPENDENCY include "../lib/Timer.mona";
3  #DEPENDENCY include "../bumperbotemergency/BumpControlES.mona";
4
5  pred bumperbotemergencyv2_BumperBotESController (
6    var2 BumperBotESController_emgStp_false,
7    var2 BumperBotESController_emgStp_true,
8    var2 BumperBotESController_bump_false,
9    var2 BumperBotESController_bump_true,
10   var2 bces_rMot_FORWARD,
11   var2 bces_rMot_BACKWARD,
12   var2 bces_rMot_STOP,
13   var2 bces_lMot_FORWARD,
14   var2 bces_lMot_BACKWARD,
15   var2 bces_lMot_STOP,
16   # time
17   var2 allTime )
18   # internal channels of component
19   = ex2 bces_tc_SINGLE_DELAY, bces_tc_DOUBLE_DELAY, bces_tc_ABORT,
        timer_ts_ALERT:
20     # defined value for all points in time
21     bces_tc_SINGLE_DELAY union bces_tc_DOUBLE_DELAY union
          bces_tc_ABORT sub allTime &
22     # unique value for all points in time
23     bces_tc_SINGLE_DELAY inter (empty union bces_tc_DOUBLE_DELAY
          union bces_tc_ABORT ) = empty &
24     bces_tc_DOUBLE_DELAY inter (empty union bces_tc_SINGLE_DELAY
          union bces_tc_ABORT ) = empty &
25     bces_tc_ABORT inter (empty union bces_tc_SINGLE_DELAY union
          bces_tc_DOUBLE_DELAY ) = empty
26     &
27     # defined value for all points in time
28     timer_ts_ALERT sub allTime &
29     # unique value for all points in time
30     timer_ts_ALERT inter (empty ) = empty
31     &
```

```
32      # parent-to-parent connectors
33      # instantiation and connection of subcomponents on channels
34      # subcomponent timer of type lib_Timer
35      lib_Timer(bces_tc_SINGLE_DELAY, bces_tc_DOUBLE_DELAY,
            bces_tc_ABORT, timer_ts_ALERT, allTime) &
36      # subcomponent bces of type bumperbotemergency_BumpControlES
37      bumperbotemergency_BumpControlES(
            BumperBotESController_emgStp_false,
            BumperBotESController_emgStp_true,
            BumperBotESController_bump_false,
            BumperBotESController_bump_true, timer_ts_ALERT,
            bces_rMot_FORWARD, bces_rMot_BACKWARD, bces_rMot_STOP,
            bces_lMot_FORWARD, bces_lMot_BACKWARD, bces_lMot_STOP,
            bces_tc_SINGLE_DELAY, bces_tc_DOUBLE_DELAY, bces_tc_ABORT,
            allTime);
```

Listing L.2: Mona translation of the composed component `BumperBotESController` shown in Listing L.1.

L.2. Example Translation of a MAA_{ts} automaton

A complete translation of the MAA_{ts} automaton inside component `ToggleSwitch` shown in Listing L.3 in MontiArcAutomaton syntax is shown in Listing L.4 in Mona syntax. The example has been used in Section 7.3.4 to illustrate the translation rules for atomic components with embedded MAA_{ts} automata.

MontiArcAutomaton

```
package lib;

component ToggleSwitch {

  port
    in Boolean pressed,
    out Boolean active;

  automaton {
    state Off, On;

    initial Off / {active = false};

    Off -> Off {pressed = false} / {active = false};
    Off -> On {pressed = true} / {active = true};
    On -> On {pressed = false} / {active = true};
    On -> Off {pressed = true} / {active = false};
  }
}
```

Listing L.3: A model of the component `ToggleSwitch` given in MontiArcAutomaton syntax.

Mona

```
# Mona predicate representing component lib_ToggleSwitch

pred lib_ToggleSwitch(
  # read values on port pressed
  var2 pressed_false,var2 pressed_true,
  # written values on port active
  var2 active_false, var2 active_true,

  # time
  var2 allTime) =
    # internal states of automaton
    ex2 On, Off:

    # defined value for all points in time
    On union Off = allTime &
    # unique value for all points in time
```

```
17      On inter (empty union Off ) = empty &
18      Off inter (empty union On ) = empty
19
20      &
21      # setting initial states and their outputs in case of strong
            causality
22      ( (0 in Off & 0 in active_false ))
23      &
24      # restriction to all points in time that we consider
25      # (to prevent infinite histories in example case)
26      all1 t: t+1 in allTime =>
27      (
28        ( t in Off
29            & (
30            t in pressed_false
31            )
32        & t+1 in Off
33        & t+1 in active_false
34        ) |
35
36        ( t in Off
37            & (
38            t in pressed_true
39            )
40        & t+1 in On
41        & t+1 in active_true
42        ) |
43
44        ( t in On
45            & (
46            t in pressed_false
47            )
48        & t+1 in On
49        & t+1 in active_true
50        ) |
51
52        ( t in On
53            & (
54            t in pressed_true
55            )
56        & t+1 in Off
57        & t+1 in active_false
58        )
59       )
60
61      # add nil completion
62      | ( ~(
63          ( t in Off
64              & (
```

```
65              t in pressed_false
66              )
67          ) |
68
69          ( t in Off
70              & (
71              t in pressed_true
72              )
73          ) |
74
75          ( t in On
76              & (
77              t in pressed_false
78              )
79          ) |
80
81          ( t in On
82              & (
83              t in pressed_true
84              )
85          )
86      )
87        # stay in same state
88        & sameNextValue(On, t) & sameNextValue(Off, t)
89        # send no message on port active
90        & t+1 notin (active_false union active_true)
91    );
```

Listing L.4: Mona translation of the component `ToggleSwitch` shown in Listing L.3.

Appendix M.

Curriculum Vitae

Jan Oliver Ringert, born August 24, 1983 in Hildesheim, Germany

Academic Employment

3/2013-11/2013	RWTH Aachen University: research and teaching assistant
4/2010-3/2013	RWTH Aachen University: fellowship of research training group Algorithmic Synthesis of Reactive and Discrete-Continuous Systems, German Research Foundation
1/2009-3/2010	RWTH Aachen University: research and teaching assistant
11/2008-12/2008	Technical University Braunschweig: research and teaching assistant
11/2004-4/2008	Technical University Braunschweig: student research and teaching assistant at the institutes: Computational Mathematics (5 semesters), Programming and Reactive Systems (1 semester), Software Systems Engineering (2 semesters)

Education

1/2009-11/2013	RWTH Aachen University: PhD studies in Software Engineering
10/2003-9/2008	Technical University Braunschweig: Computer Science studies; diploma in Computer Science with distinction
06/2003	Scharnhorstgymnasium Hildesheim: German Abitur
8/2000-7/2001	Connell High School, Connell, WA, US: student exchange year

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] we show how this looks like and how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P described in the two books [Rum11] and [Rum12] are implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams (ADs) [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02] The question how to adapt and extend the UML in discussed in [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99], [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06], [KRV10], [Kra10] describes an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools

can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been examined in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory. [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied on class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] embodies the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development, maintenance and [LRSS10] technologies for evolving models within a language and across languages and linking architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Many products exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures the commonalities as well as the differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are added (that sometimes also modify the core). A set of applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. And we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specifications concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b] that perfectly fits Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development. Application classes like Cyber-Physical Systems [KRS12], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools. We tackle these challenges by perusing a model-based, generative approach [PR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g.,c collecting data from temperature, cars etc. are easily developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, October 2007.

[BCGR09a] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, 2009.

[BCR07a] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, February 2007.

[BCR07b] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, TU Munich, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In M. Broy and B. Rumpe, editors, *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, LNCS 1526, pages 43–68, Bernried, Germany, October 1998. Springer.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Proceedings of the 10th Workshop on Automotive Software Engineering (ASE 2012)*, pages 789–798, Braunschweig, Germany, September 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*. Springer, 2012.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, CfG Fakultät, TU Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, LNCS 5795, pages 670–684, Denver, Colorado, USA, October 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-Based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO' 11)*, New York City, USA, October 2011.

[FPPR12] Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Proceedings of the 7th International Conference on Energy Efficiency in Commercial Buildings (IEECB)*, Frankfurt a. M., Germany, April 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, Paderborn, Germany, October 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, Toulouse, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Informatik Bericht 2008-01, pages 76–89, CFG Fakultät, TU Braunschweig, March 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TUM, Munich, Germany, 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Technical Report 2006-04, CfG Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings der Modellierung 2006*, Lecture Notes in Informatics LNI P-82, Innsbruck, März 2006. GI-Edition.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TUM, Munich, Germany, 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. 16th Monterey Workshop*, LNCS 6662, pages 17–32, Redmond, Microsoft Research, 2011. Springer.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality. 18th International Working Conference, Proceedings, REFSQ 2012*, Essen, Germany, March 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Model Driven Engineering Languages and Systems, Proceedings of MODELS*, LNCS 6394, Oslo, Norway, 2010. Springer.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo*, pages 22–31. ACM, September 2013.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab / Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 11–18, New York, NY, USA, 2013. ACM.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, pages 99–113. Springer, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineeering in Research and Practice*, Las Vegas, Nevada, USA, 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI) at ICSE 2012*, pages 61–66, Zurich, Switzerland, June 2012. IEEE.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, Oct 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of International Software Product Lines Conference (SPLC 2011)*. IEEE Computer Society, August 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH*, February 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208, Oxford, UK, March 2012. Springer.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik in Berlin*, Lecture Notes in Informatics LNI 198, pages 181–192, 27. Februar - 2. März 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.)*, Techreport B-108, Helsinki School of Economics, Orlando, Florida, USA, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop MOTPW'12, Innsbruck, October 2012*, pages 2:1–2:6. ACM Digital Library, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. P. Dini, IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag Aachen, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Germany, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 323–338. Chapmann & Hall, 1996.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM'06), Portland, Oregon USA*, Technical Report TR-37, pages 150–158, Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07), Montreal, Quebec, Canada*, Technical Report TR-38, pages 8–10, Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, October 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In R. F. Paige and B. Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Zurich, Switzerland, 2008*, Lecture Notes in Business Information Processing LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 241–270. Springer, October 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proc. Euro. Soft. Eng. Conf. and SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 592–607, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Proc. 25th Euro. Conf. on Object Oriented Programming (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 153–167. Springer, 2011.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In G. J. Chastek, editor, *Software Product Lines - Second International Conference, SPLC 2*, LNCS 2379, pages 188–197, San Diego, 2002. Springer.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, LNCS 873. Springer, October 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In J. Davies J. M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In H. Kilov and K. Baclawski, editors, *Practical foundations of business and system specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[PR13] Antonio Navarro Perez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proc. of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. Co-located with MODELS 2013, Miami*, Sun SITE Central Europe Workshop Proceedings CEUR 1118, pages 15–24. CEUR-WS.org, 2013.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. *Software Engineering 2013 Workshopband*, LNI P-215:155–170, May 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), May 6-10, 2013, Karlsruhe, Germany*, pages 10–12, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, Hershey, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever, editor, *Formal Methods for Components and Objects*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future. 9th International Workshop, RISSEF 2002. Venice, Italy, October 2002*, LNCS 2941. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, second edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, second edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Germany, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 57–76, October 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, Karlsruhe, Germany, May 2013. IEEE.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Germany, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Germany, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In D. Schaefer, editor, *Proceedings of the SESAR Innovation Days*. EUROCONTROL, November 2011.