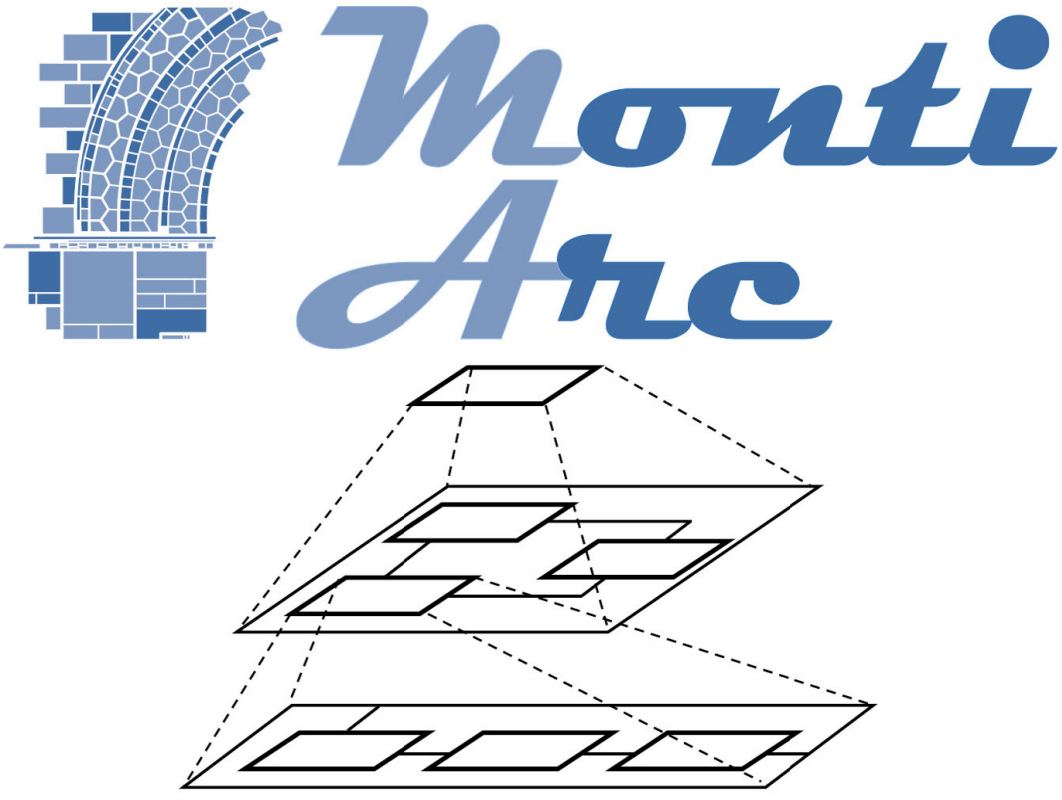
SE | RWTH AACHEN UNIVERSITY

Arne Haber

MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems



Aachener Informatik-Berichte,
Software Engineering

Band 24

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

MontiArc - Architectural Modeling and Simulation of Interactive Distributed 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.-Wirt.-Inf.
Arne Haber
aus Wolfsburg

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessorin Dr.-Ing. Ina Schaefer

Tag der mündlichen Prüfung: 22. März 2016

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8440-4697-7 erschienen.

[Hab16] A. Haber:
MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.
Shaker Verlag, ISBN 978-3-8440-4697-7. Aachener Informatik-Berichte, Software Engineering, Band 24. 2016.
www.se-rwth.de/publications/

Abstract

Software architecture is the essence of a system and determines important functional as well as non-functional properties [BCK03]. It decomposes the system into small, manageable subsystems respectively components that interact over well defined interfaces. Formal architecture description languages (ADLs) offer great potential to model and analyse the architecture of a system, predict the overall performance of a system using simulations, and even allow to automatically generate parts of the implementation.

Nevertheless, ADLs are rather not used in industrial practice since several problems of architectural modeling hinder to exploit their potential to the full extend. Either an ADL is too general to provide enough information for formal analyses, or it is tailored well to a specific domain and development process, so it cannot be easily applied to another context. Beside language barriers caused by uncommon languages, most ADLs are regarded to be complex and heavyweight [MLM+13]. Good modeling tools are missing [Pan10] and existing tools cannot be easily integrated into existing tool chains [WH05]. Also, incremental modeling and model reuse is most often not supported.

This thesis elaborates the design of an ADL that copes with these impediments of architectural modeling in practice. Therefore, the design of a lightweight and easy to learn ADL is derived which also provides well defined extension points to be adapted to a certain domain or development process. Controlled reuse of architectural models is explored. Furthermore, it is investigated how architectural modeling can be enriched with agile development methods to support incremental modeling and the validation of system architectures.

Therefore, a detailed set of requirements for architectural modeling and the simulation of system architectures is defined. Based on these requirements, MontiArc, a concrete ADL to model logical architectures of distributed, interactive systems, is derived. The language is based on the mathematical FOCUS [BS01] framework, which allows to simulate modeled systems in an event-based style. Code generators and a simulation framework provide means to continuously refine and test architectural models.

The language and the corresponding tools provide extension points to easily add new features to the language or even adapt it to a new domain. For this purpose, a corresponding language extension method is presented to extend the syntax, language processing tools, and code generators of the ADL. Furthermore, a lightweight model library concept is presented which allows to develop and reuse component models and their implementation in a controlled and transparent way. The developed language, the simulator, and the language extension techniques have been examined in several case studies which either used or extended MontiArc.

Danksagung

An dieser Stelle möchte ich mich herzlich bei den Menschen bedanken, die zur Fertigstellung und zum Gelingen dieser Dissertation beigetragen haben.

Besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Betreuung dieser Dissertation. Durch zahlreiche konstruktive Diskussionen mit Bernhard und seine wertvollen Ratschläge hat er entscheidend zum Gelingen der Promotion beigetragen. Ebenfalls bedanken möchte ich mich für die Möglichkeit in vielen Industrie- und Forschungsprojekten Erfahrungen außerhalb des akademischen Elfenbeinturms zu sammeln. Diese Erfahrungen haben maßgeblich zu meinem weiteren Werdegang beigetragen.

Weiterer Dank gebührt Prof. Dr.-Ing. Ina Scheafer für die gemeinsamen spannenden Arbeiten und Projekte im Bereich der Varianten- und Deltamodellierung. Über ihre Bereitschaft meine Dissertation als Zweitgutachterin zu betreuen, habe ich mich besonders gefreut. Prof. Dr. Ir. Joost-Pieter Katoen möchte ich für die Leitung der Promotionskommission sowie für die Abnahme der Prüfung in der theoretischen Informatik danken. Herrn Prof. Dr. Stefan Decker danke ich für die Bereitschaft mich in der praktischen Informatik zu prüfen.

Natürlich möchte ich mich sehr herzlich bei meinen Kollegen bedanken, mit denen ich die Zeit am Lehrstuhl, beim Kickerspielen oder bei abendlicher Zerstreuung verbracht habe. Ihr habt dazu beigetragen, dass Aachen ein Stückchen Heimat für mich wurde, auch wenn ich dem Karneval nie etwas abgewinnen konnte. Für das Korrekturlesen früher Fassungen der Dissertation bedanke ich mich bei Lars Hermerschmidt, Andreas Horst, Thomas Kurpick, Markus Look, Klaus Müller, Pedram Mir Seyed Nazari, Antonio Navarro Pérez und Andreas Wortmann. Da sich die entsprechenden Unterlagen bereits im Keller befinden (wer eine Dissertation geschrieben hat, kann das sicherlich nachvollziehen), habe ich hier hoffentlich niemanden vergessen. Für das Anhören des Probevortrags und hilfreiches Feedback möchte ich Achim Lindt, Markus Look und Andreas Wortmann danken.

Dr. Jan Oliver Ringert danke ich für die gemeinsame Arbeit am MontiArc Sprachdesign und dem technischen Bericht. Darüber hinaus nochmal vielen Dank an die “Technologiesau” Antonio und Andreas H. für euren stetigen Kampf für Maven und die Unterstützung bei der Umstellung von MontiArc auf dieses Buildsystem. Dr. Holger Krahn, Dr. Steven Völkel und Dr. Martin Schindler gilt mein Dank für ihre Arbeiten an MontiCore, auf deren Basis meine Arbeit aufgesetzt hat. Marita Breuer und Galina Volkova danke ich für den technischen MontiCore-Support. Bei Sylvia Gunder und (Dr.) Holger Rendel bedanke ich mich für die Unterstützung bei der organisatorischen Vorbereitung von Holgers und meiner Promotionsprüfungen, die am selben Tag durchgeführt wurden.

Bedanken möchte ich mich ebenfalls bei den vielen Studenten, die mich in ihrer Abschlussarbeit oder als Hiwi bei der Implementierung in und um MontiArc oder bei Projekten, unterstützt haben. Hierzu gehören: Markus Arndt, Paul Chomicz, Christoph Hommelsheim, Tim Ix, Oliver

Kautz, Alexander Kogaj, Thomas Kutz, Juha Veikko Lauttamus, mein späterer Kollege Pedram Mir Seyed Nazari, Rajeevan Rabindran, Sebastian Roidl, Stefan Schubert und Minh Quan Tran. Bei Claas Oppitz bedanke ich mich für das Design und die Erstellung des MontiArc Logos.

Abschließend möchte ich meinen Eltern und meiner Familie für die Unterstützung auf meinem Lebensweg danken. Ihr habt mir das Studium und die anschließende Promotion ermöglicht und mich stets in meinem Vorhaben bestärkt. Besonderer Dank gebührt natürlich Nathalie, die mich nicht nur unterstützt, sondern auch meine Laune bei der Erstellung dieser Arbeit aufgeheitert hat. Danke, dass Du gemeinsam mit mir für die Zeit der Promotion nach Aachen gezogen bist. Ohne Dich wäre diese Arbeit nicht möglich gewesen.

Braunschweig, Juli 2016
Arne Haber

Contents

Chapter 1

Introduction

Modern software systems are becoming more and more complex. For example, modern car systems contain up to 80 interacting control devices which realize advanced features, such as lane keeping assistants, automatic parking, and adaptive cruise control, whose functionalities are mostly provided by software [RSG+08]. In interactive systems, complex functions are often realized by cooperating components. Such components can include and depend on software which is running on control devices and hardware such as sensors and actuators [BS01]. Considering that components are used in different car series, evolve during the life-cycles of the series, and have to be tailored to dedicated variants, ensuring the compatibility of components and integrating them to complete system is an inherently complex task.

Of course, a complex software system requires a software development process which is able to handle this complexity. During the last decades, many approaches have been developed to manage the rising complexity of software systems. Most traditional development processes, such as the waterfall model [Roy70], the spiral model [Boe88], or the V-model XT [BR05, FHKS09], have in common, that the process is separated into distinct phases. In the design phase, the structure of the system under development (SUD) is defined in terms of a software architecture. According to Bass et al. [BCK03], the software architecture is very important due to its three main cases of application: communication among stakeholders, early design decisions, and a transferable abstraction of the SUD.

The software architecture of a SUD decomposes the system into small, manageable subsystems respectively components that interact over well defined interfaces. This explicit definition of interfaces between components effects the system development in several positive ways. It facilitates reuse of development artifacts and methods since their interfaces serve as contracts between the components and their user. Controlled reuse of well tested components lead to reduced costs, a decreased time to market, and improved quality [Lim94, MBB95]. Also architectural patterns can be reused, which provide widely used and accepted solutions for recurring problems during the architectural system design [FK05].

Further, the hierarchical decomposition of a system into components with well defined interfaces also decomposes the complex system development task into a set of smaller and manageable component development tasks. Consequently, these components can be developed independently in concurrent activities which reduces the overall time to market.

Software architecture is the essence of a system and determines important functional as well as non-functional properties [BCK03]. Consequently, according to Garlan et al. [GMW97], a well defined architecture can lead to a product that satisfies the given requirements, while an inappropriate architecture can be devastating. Depending on the development process, architectural

descriptions can be used for communication or constructively, e.g., in the Model-Driven Architecture (MDA) approach [MSUW02, KM05], to generate implementation parts of the SUD.

To avoid misunderstandings in the communication between developers and to make architectural descriptions available for automatic processing, a well defined modeling language is needed which allows to describe SUDs and their configuration in a meaningful way. These languages, commonly known as architecture description languages (ADLs), allow a high level description of systems with a well defined and precise textual or graphical notation. An architectural description not only defines the structure of a system but also includes its behavior and patterns of interaction [BCK03]. In this way, ADLs also enable reasoning about specific system properties in early development stages [GMW97].

The topics of this thesis are architectural modeling and ADL engineering. To give a common understanding of the used terms, they are introduced in the following section. Afterwards, the thesis is motivated in Section 1.2 and the concrete context is described in Section 1.3. Then, the objectives are described and the main results are concluded. Finally, the structure of the thesis is outlined in Section 1.6.

1.1 Terms and Definitions

In the following, a list of terms and definitions is given that are used in this thesis.

- The **Software Architecture** of a system is the structural description of elements from which the system is built and the interaction among the contained elements. Patterns in the defined architecture and externally visible properties of architectural elements impose constraints on how elements of a system can be composed and connected [SG96, BCK03].
- The **Logical Architecture** of a system is a logical viewpoint on a system. In contrast to the structural focus of a software architecture, which exclusively contains software elements of a system, logical architectures describe logical functions and their interaction. These functions, which may be realized by hardware or software deployed on different hardware, must cooperate to achieve the common task of the system [GHK+07]. Consequently, aspects of deployment and distribution are not part of the logical architecture.
- An **Architecture Description Language (ADL)** provides means to describe the software architecture of a system with
  - components as architectural elements,
  - component interfaces, which define the externally visible parts of a component,
  - connectors, which model the interaction between components, and
  - the architectural configuration of a system, which defines the structural composition of elements [AAG93, MT00].

  Beside the concrete syntax that is used to describe elements of a system and their interaction, an ADL provides a conceptual framework [GMW97] which reflects characteristics of the ADL's application domain and the corresponding architectural style [MT00].
- Since systems from different domains have fundamental differences in their structure and communication, **Architectural Styles** are used to tailor an ADL to a specific application domain [SG96, MT00, DvdHT01]. For example, the architecture of an enterprise infor-

mation system cannot be described with the same means as the architecture of an embedded, interactive system. The fundamental syntax of ADLs usually comprises boxes, to model components, and lines, to express communication between components. The architectural style of an ADL assigns a specific semantics to these model elements which allows to interpret architectural models in a uniform and unambiguous way [AAG93]. Beside the structural definition, an architectural style inheres a communication semantics that describes patterns of runtime control and data transfer. In particular, it describes how components perform computations and which kind of communication is modeled with connectors. Further, an architectural style can impose constraints to the underlying meta-model which determine semantically valid models.

- A **Component** is a unit of computation or data storage which interacts with its environment via well defined interfaces [MT00]. It accomplishes its tasks with internal computations and external communication with other components [AAG93]. A component may scale from a single procedure to an entire application or subsystems and usually corresponds to a compilation unit in the modeled system [MT00].
- A **Component Interface** defines a set of interaction points between the component and its context as externally visible properties. It specifies the services which a component provides and the services which are required by the component to fulfill its functionality. Most often, services are messages, operations, or variables [MT00].
- **Connectors** are used to model inter-component communication. Depending on the ADL, connectors are implicitly (in the architectural style) or explicitly (as first level elements) associated with a set of rules that govern those interactions. Depending on those rules, connectors can represent complex interaction patterns, such as communication protocols, and communication styles, such as message passing or function calls [MT00].
- An **Architectural Configuration** describes the topology respectively decomposition of a system or component as a connected graph of components and connectors [AAG93, MT00]. This structural definition is used to determine whether connected components are compatible and their interfaces match. Further, architectural analyses are possible for adherence to design heuristics, metrics, and architectural style constraints [MT00]. This thesis distinguishes between **Component Type Definitions** (or short **Components**, see above) and **Subcomponent Declarations** (or short **Subcomponents**). The former define the interface and architectural configuration of a component, the latter instantiate a component type definition as an element of another component type definition.
- A **Decomposed Component** is a component with a given architectural configuration. The behavior of a decomposed component is rendered by the composed behavior of the contained subcomponents.
- An **Atomic Component** is not decomposed. It rather has a corresponding implementation that renders the behavior of the component.
- An **Architectural Programming Language (APL)** introduces architectural modeling elements, i.e., components, interfaces, connectors, and architectural configuration, into a general purpose language (GPL) [BHH+06]. Thus, it can be regarded as an ADL combined with programming concepts to implement component behavior. The main goals of architectural programming are: First, architectural design decisions are explicitly present

in the implementation and the architectural design is enforced. Second, architectural erosion of the implementation, which is caused by software evolution [PW92], is encountered since architecture and implementation consistently evolve [ACN02b].

1.2 Motivation

Development processes, as already mentioned, structure the software development into distinct phases to increase controllability and to improve the predictability of the development process. For example, a well established process in the domain of interactive systems, e.g., in the automotive domain, is the V-Model that logically arranges activities in form of a "V". The left branch of the V contains successive design and development activities: requirements engineering, system design, software architecture design, software component design, and the implementation phase at the bottom of the V. The right branch contains corresponding test activities: unit, integration, system, and acceptance testing. Since the architectural design is located early in the sketched development process, it has a huge impact on the SUD and the following development activities. An inappropriate architectural design is repeated in the implementation and can be initially discovered by system and acceptance testing activities. Garlan et al. [GMW97] even consider architecture as the essence of a system which significantly influences whether the given requirements are met or not.

Several studies summarized by Galin [Gal04] and Feiler [Fei14] outline that 50 % of the defects in software systems are introduced by invalid requirements, an inaccurate system respectively architectural design. 35 % of these defects are detected during the integration and system testing. 15 % are not discovered until a product is released. Since many activities in the development process have to be repeated, correcting these defects causes high costs. According to an IBM white paper presented by Briski et al. [BCH+08], the costs to fix defects of an already released product are up to 30 times higher than if the same defect would have been fixed in the design phase of the development. Galin [Gal04] even reports an average relative defect cost of 40 times if a defect is detected during system integration tests. It is even 110 times higher if it is detected in the operation phase of a software product.

As a consequence, the early validation of software architectures, automatic code generation from logical and physical ADLs to enforce architectural properties within the system implementation, as well as controlled reuse of well defined architectural components yield great potential to decrease development costs and improve the quality of a system. Nevertheless, several problems of architectural modeling in practice hinder to exploit this potential to the full extent:

- In practice, architectures are most often modeled using general purpose languages, such as the Unified Modeling Language (UML), which allow to design the architecture and communicate it to the developers [MLM+13]. Since the underlying architectural style is rather general, constructive use, e.g., for code generation, analyses of system properties, impact, or dependency analyses, is not automatically applicable for such languages.
- The software architecture of a system is designed initially and, at best, the implementation is created in conformance to the initial design. Thus, the architecture is also present in the code [TMD09], even if an explicit mapping is missing. Discovered defects, changing requirements, or the evolution of a software system might lead to changes in the architec-

ture. Due to hard development deadlines, these architectural changes are directly realized in the code and not reflected back into the architectural model, which leads to architectural erosion [PW92].

- Architectural erosion may further lead to incompatible interfaces of component implementations. If continuous integration services are not available, which immediately integrate changed component implementations with their environment, these incompatibilities are revealed when manually integrating the system. Consequently, the integration effort is hard to predict.
- Software architecture connects system requirements with the concrete implementation [GHK+07]. If the used modeling tool does not support tracing from requirements to architecture to code and vice versa, these relations have to be documented externally, e.g., in formal review documents. Caused by architectural erosion, these manually created mappings are blurred over time.

1.3 Context

In practice, distinct system kinds and domains require different means and notations for architectural modeling. This thesis focuses on architectural modeling of interactive systems. Since the focus is set on the logical software architecture of such systems, the concrete deployment of components and aspects of the involved hardware are not considered. According to Broy and Stølen [BS01], an interactive system consists of distributed components which communicate with asynchronous message passing over buffered and directed channels. Due to the underlying distribution, components cannot implicitly share their state and do not access shared data synchronously. Since no effort is needed to synchronize data access, this architectural style also eases the development of parallel, non-distributed systems. Generally, the software part of Cyber Physical Systems [Lee10] can be considered as an interactive system [TMD09].

Typical examples for interactive systems are:

- telecommunication systems,
- distributed business applications based on a service oriented architecture or in the cloud,
- multi-core architectures, and
- control devices in automotive systems or manufacturing lines.

A more concrete example for an interactive system taken from the automotive domain is given in Figure 1.1. It depicts the logical architecture of component `LightCtrl`, the interior light control of a car. The interface of the component is given by three incoming ports on the left side and one outgoing port on the right side. All depicted ports have a type which denotes the type of the message that can be received or sent. In the used graphical notation, the message type of the ports is annotated on the outgoing or incoming connectors.

The architectural configuration of component `LightCtrl` is given by a set of subcomponents that instantiate other component definitions. In this way, the component definitions `Arbiter`, `DoorEval`, and `AlarmCheck` are reused as subcomponents. The architectural configuration is completed by connectors which connect ports of `LigthCtrl`'s interface with ports provided by the interfaces of the subcomponents. Also subcomponents can be directly connected, as for example subcomponent `DoorEval` and `Arbiter`. Since the model is asso-

ciated with the aforementioned architectural style of interactive systems, all depicted connectors model asynchronous message passing between the depicted components.

1.4 Objectives

Current ADLs for the modeling of interactive systems, which focus on the problems described in Section 1.2, are rarely used in industrial practice. A survey based on interviews of 48 practitioners from 40 different IT companies presented by Malavolta et al. [MLM+13] revealed, that most often generic UML tools are used in industrial architectural modeling. This mainly is caused by the following reasons:

- Language barrier: UML is a widely known modeling language which is taught to many engineers.
- Complexity: Most ADLs are regarded to be complex and heavyweight, which deters practitioners to use them [MLM+13].
- Modeling tools: Only few ADLs are supported by satisfactory tools [Pan10].
- Tool integration: If ADL tools exist, they are not (well) integrated into existing desktop environments that are already in use [WH05].
- Incremental modeling: ADLs do not provide support for incremental adaption since they are not well integrated into the development process respectively the used tool chains [Woo05]. Furthermore, a modular design of component models with loose coupling to other component models is a precondition for incremental modeling, analyses, and code generation.
- Unspecific architectural style: Some ADLs are too generic and lack domain specific entities which are important to architects [WH05]. Since they are not tailored to a certain architectural style, they do not provide enough information for automatic analyses or code generation. Thus, they need to be extended, which may result in ambiguous or unclear semantics.
- Fixed architectural style: Other ADLs often make restrictive assumptions and impose a particular architectural style. These assumptions may be (slightly) inappropriate for a specific use case [Pan10]. If an ADL cannot be easily adjusted to a current use case, it

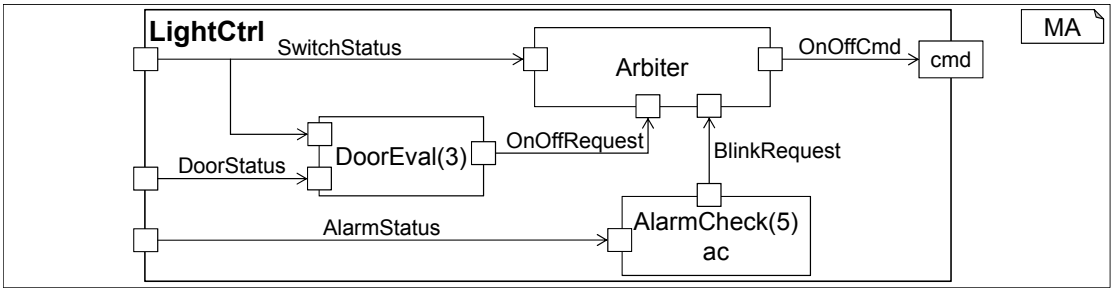


Figure 1.1: Component type definition `LightCtrl` that defines the architecture of the interior light control of a car.

cannot be used. Furthermore, it is not economically efficient to learn a distinct ADLs for each needed architectural style.

- Model reuse: While architectural modeling eases reuse of developed software components, the architectural models are most often not reusable in a controlled manner [Woo05].

This thesis addresses the question, how to design an ADL which copes with the aforementioned impediments for architectural modeling in practice. An ADL for the modeling of logical architectures of interactive systems is presented. To efficiently support the system development within early development phases, it allows to analyze, explore, and validate the SUD in an agile way. Therefore, a simulation has been constructed which allows to validate a system and predict its properties. The developed language and corresponding tools serve as a case study to validate the derived concepts of architectural modeling.

Both, the architectural style of the ADL and the corresponding simulation are based on a formal semantics named FOCUS [BDD$^+$93, BS01]. FOCUS provides several advantages:

- System and component specifications with a precise semantics expressed in predicate logic.
- A development process for interactive systems based on controlled and verifiable system refinement.
- A formal model of communication that allows to describe and reason about communication channels and event traces.
- Mathematical foundations to describe time within the communication model.

Due to this formal foundations, the developed architectural style already provides enough information to analyze and simulate interactive systems. Nevertheless, it is also possible to extend and refine the developed language and tooling to tailor the ADL to specific needs. In this way, concrete implementation languages or new modeling elements can be introduced into the language easily.

Admittedly, FOCUS provides a well-defined semantics for architectural modeling. Nevertheless, since it is a formal and theoretical framework, a workbench is needed to use it in practice and benefit from its advantages. To avoid the aforementioned drawbacks of heavy-weight ADL tools, this thesis further poses the question, how architectural modeling in early development phases can be combined with agile methods. Thus, the developed language is inspired by agile methods which heavily depend on automatically executed tests. By directly implementing the behavior of components in an asynchronous, event-based way, rapid prototyping of interactive software architectures is simplified. These models can be automatically analyzed and validated with simulations without user interaction, which simplifies distributed development of architectures. Since architectural models are first-level artifacts, they are continuously aligned with the simulation code, which prevents architectural erosion.

Another important aspect of the developed ADL is reuse. It provides syntactical elements which increase the reusability of components. Similar to Java's jar archives, components and the corresponding simulation code can be packaged together in component libraries, which can be easily reused in other development projects.

Summing up, this thesis poses the research question: How to design an ADL that copes with the impediments of architectural modeling in practice? This question is refined into the following subquestions:

RQ1 How to design a lightweight and easy to learn ADL?

RQ2 How to design an extendable ADL which allows to reuse as much tooling as possible?

RQ3 Which concepts can be applied to an ADL to support reuse of architectural models?

RQ4 How to integrate agile development methods with architectural modeling to allow for incremental modeling and early validation of the architecture?

1.5 Main Results

The main results of this these are:

- A concrete ADL to model logical architectures of distributed, interactive systems. Its architectural style is based on the mathematical FOCUS framework.
- A FOCUS based simulation of ADL models that allows to validate and explore the interaction of system components.
- An Eclipse-based integrated development environment (IDE) to simplify modeling.
- A structured ADL extension method to refine and adjust the given architectural style.
- Language processing tools extendable and adjustable to be reused for extended languages. This comprises a symbol table, context conditions for model analysis, a transformation framework, and code generators for architectural simulation. In this way, analysis for customized languages can be developed easily and the simulation can be tailored to also interpret newly added elements.
- Finally, the developed language can also be regarded as a case study for the compositional development of languages and tools [Völ11, HLMSN+15].

1.6 Thesis' Structure

The thesis is structured as follows:

- Chapter 2 defines important requirements for architectural modeling and simulation. Further, existing ADLs are reviewed regarding these requirements.
- Chapter 3 presents the developed ADL named MontiArc and its architectural style. Beside its defining grammars, the context conditions are presented which are used for model analysis. Finally, the compatibility of MontiArc and the Architecture Analysis and Design Language (AADL) [FGH06], one of the mostly used ADLs in industrial practice, is discussed.
- Chapter 4 presents the MontiArc simulation. For this, the needed FOCUS foundations are discussed, the scheduling approach, the runtime environment (RTE), and supported timing domains are presented.
- Chapter 5 documents the technical realization of MontiArc. This comprises aspects of model processing, the symbol table as the foundation of an extendable language, provided transformations and generators. Further, the mechanisms to implement and integrate atomic component implementations, and the available MontiArc modeling tools are presented.

- Chapter 6 contains a tutorial that explains how to model and simulate interactive systems using MontiArc. This comprises: modeling, behavior implementation, validation and optimization testing, documentation, library development, and the distribution of simulations.
- Chapter 7 presents a structured method for the extension of MontiArc. It allows to extend the model processing to develop new analysis, metrics, and transformations. Further, extension methods for the simulation are presented. Finally, a method to extend the syntax of MontiArc is given.
- Chapter 8 presents selected case studies in which MontiArc has been used to simulate software architectures.
- Chapter 9 gives an overview of selected languages which extended MontiArc.
- Chapter 10 contains a discussion whether the defined language and simulation requirements are fulfilled by MontiArc and the corresponding modeling tools. Further, the thesis is concluded and outlook for possible future work is given.

Chapter 2

Requirements for Architectural Modeling and Simulation

The software architecture of a system determines important quality attributes such as extensibility, robustness, and fault tolerance [BCK03]. Architectural modeling with architecture description languages (ADLs), which have a well defined architectural style, offers the possibility to design the system under development (SUD) in an unambiguous way. Further, automatic analyses of important system properties in an early development phase and code generation are made possible. Nevertheless, as discussed in Chapter 1, ADLs are rather not used in industrial practice due to missing language features or unpractical handling of the corresponding tools. A recent study in industry by Malavolta et al. [MLM+13] identified a well-defined semantics, tool support, analysis, versioning, collaborative development, and extensibility among the most useful and desirable features of an ADL. Since these features are most often missing in existing ADLs, rather general purpose modeling languages, e.g., the Unified Modeling Language (UML), are used for architectural modeling.

The posed research questions and listed ADL features are broken down into requirements which determine important properties of the ADL in Section 2.1 and requirements for the simulation of architectural models in Section 2.2. The chapter concludes with an overview of existing ADLs and corresponding simulation approaches.

2.1 Requirements for Architectural Modeling

The following list of language requirements (LRQs) defines important properties of ADLs for interactive systems. The defined properties are derived from the posed research question *RQ1*, *RQ2*, and *RQ3*. Hence, they are inspired by the aforementioned impediments for using ADLs in practice (see Section 1.4). The LRQs also define a basic architectural style that is suitable for the modeling of interactive systems.

LRQ1 **Architectural Style**: To be well suited for modeling of interactive systems and for rapidly prototyping a SUD, a core architectural style has to provide means for modeling of the most important artifacts of interactive systems. To be easy to use, it is necessary to not overload an ADL with a rich set of specialized model elements. Nevertheless, an architectural model has to provide enough information to analyze and validate modeled systems. In accordance to the classification of Medvidovic and Taylor [MT00], mandatory ADL elements are: component definitions, their interfaces, connectors, and

the architectural configuration of components to define the composition of a system. The following LRQs further define important properties of these architectural elements.

LRQ1.1 Components: Component definitions have to fulfill the following properties:

LRQ1.1.1 Component Reuse: Modeled components have to be reusable in different contexts. Therefore, it is necessary to provide suitable techniques which allow to instantiate existing component definitions and to adjust them in a controlled way. Furthermore, it is necessary to provide techniques for defining components based on existing component definitions without replicating the model.

LRQ1.1.2 Packages and Compilation Units: To support collaborative development and reuse of components, it is necessary to unambiguously identify components and the location of the corresponding defining model (compilation unit).

LRQ1.1.3 Timing Classification: A classification of component timings is necessary to analyze and simulate the behavior of complex composed components. A component belongs to a *timing domain* which defines the component's awareness of time and how it processes messages.

LRQ1.1.4 Inner Component Definitions: Sometimes, components are not intended for reuse but are rather used to structure big components into manageable parts. To easily model architectures with singleton-alike components, it is necessary to provide modeling techniques for the definition of inner components that are only visible within the component in which they are defined. Further modeling techniques are nececarry to simplify the instantiation of inner components in a controllable way.

LRQ1.1.5 No Shared State: A component encapsulates its internal state and does not share it directly with other components. If shared information is needed, components explicitly synchronize by exchanging messages.

LRQ1.2 Interface: Interactive systems communicate with asynchronous messages that are transmitted over typed channels. Thus, the interface of an interactive component, which defines a set of typed incoming and outgoing ports, can serve as connection points for channels. Furthermore, concepts for adapting a component to the current context (see requirement *LRQ1.1.1*) need to be part of a component's interface. Since they have to be used when a component is instantiated, they cannot be encapsulated into the component's internal representation.

LRQ1.3 Architectural Configuration: An ADL has to provide means to define the architectural configuration of a component. This comprises the internal structure of a component and communication paths within this structure. The architectural configuration is not part of a component's interface. Thus, it shall not be accessible for other components. Consequently, decomposed and atomic components need not be distinguished when they are instantiated as a subcomponent or as a system.

LRQ1.4 Synonymously Used Type and Name: In practice, static architectures rarely contain multiple instances of the same component. For example, in automotive systems developed with Matlab/Simulink, a subsystem block represents type and instance of a component type. Thus, type and name of a component are often used as

synonyms. Consequently, a port or a subcomponent in a component type definition with a unique type (its type is only used once) does not need an explicit name.

LRQ1.5 **Autoconnect**: Experiences have shown that sender and receiver of a connector often have the same port name in practice. If these names are unique, the connector can be derived automatically. If autoconnect is active, it is necessary to inform the modeler about derived connection.

LRQ1.6 **Constraints**: Component constraints are useful to restrict the valid behavior or state of components on the architectural layer. For an extendable ADL (see requirement *LRQ3*), a flexible integration mechanism is necessary to integrate different constraint definition languages.

LRQ1.7 **Documentation**: To simplify reuse of component definitions, it is necessary to document their interface and provided behavior. An in place documentation, which is tightly coupled to the component model, supports collaborative development and is more likely kept in sync with the component.

LRQ1.8 **Compatibility to AADL**: The Architecture Analysis and Design Language (AADL) is one of the most popular ADLs in industrial practice [MLM+13]. To be able to reuse existing analyses and tools of the AADL, the essential core concepts have to be compatible to a selected set of AADL model elements and vice versa.

LRQ2 **Usability**: ADLs and architectural programming languages (APLs) have to be easily usable. This comprises the following aspects of tool support and collaborative development:

LRQ2.1 **Tool Infrastructure**: To support agile component development, a tool infrastructure which allows to process component models without user interaction is necessary. This comprises parsing, symbol table construction, and context condition checking (language front end), as well as simulation code generation (language back end) and test execution.

LRQ2.2 **IDE**: An integrated development environment (IDE) is crucial for the acceptance of a programming or modeling language. A text editor with syntax highlighting, an outline, and active specification with auto-completion supports the modeler to create valid models. Integrated component documentation eases reuse of library components.

LRQ2.3 **Context Conditions**: Context conditions of an ADL define modeling constraints of the associated architectural style. They define which models are well-formed. This comprises the correct usage of connections as well as the integrity of references within a model. Automatically validated context conditions are necessary to support distributed, agile component development. Therefore, context condition checks need to be integrated in the IDE (see requirement *LRQ2.2*).

LRQ2.4 **Wizard**: To quickly get started, a project wizard is necessary that sets up a development project for architectural modeling.

LRQ2.5 **Distributed Development**: To support distributed development of architectural models, it is necessary to integrated distributed revision control clients into the IDE. Further, it is crucial to be able to merge different development versions of a model and automatically validate if current changes render a valid model.

LRQ2.6 Tutorial: To introduce developers to an ADL and to guide them through the development of components, a tutorial is necessary. This especially includes how to:

- technically install the ADL IDE (see requirement *LRQ2.2*),
- configure the build infrastructure (see requirement *LRQ2.1*),
- model an illustrative example,
- implement behavior of atomic components (see requirement *SRQ2*),
- validate the interface behavior of components and signal flows within components (see requirement *SRQ7*),
- document components (see requirement *LRQ1.7*), and
- create and use component libraries (see requirement *LRQ5*).

LRQ3 Reusability and Extensibility: An impediment for the use of ADLs in practice is that they either have an architectural style which is too generic [WH05] or have a fixed architectural style [Pan10]. Consequently, the architectural style of an ADL needs to be adjustable to tailor the ADL to specific needs or domains. This comprises the following requirements:

LRQ3.1 Add or Refine Model Elements: To adjust an ADL to a certain domain, e.g., architectures of robotics or cloud applications, it is necessary to add new kinds of architectural elements and thus extended the language. In addition, it further has to be possible to refine or replace language elements.

LRQ3.2 Behavior Description Extension Point: Components do not necessarily contain constructive definitions of their behavior. To allow the uniform integration of abstract behavior descriptions, e.g., specification automata, or concrete behavior implementations given in general purpose language (GPL) code, a predefined extension point is suitable.

LRQ3.3 Modularity: To allow for reuse of language processing tools within the tooling of an extended language, it is necessary to develop these tools in a modular way. This comprises context conditions, the symbol table, transformations, and code generators.

LRQ4 Type System: A type system is required to instantiate component and data type definitions and is thus a precondition for reuse. Further, an important feature of ADLs is the analysis of modeled systems [MT00]. To support static validations of architectural models, a type system is needed which comprises strongly typed data types, ports, and components. Type checking of architectures then ensures the consistency of a model. The following properties of type systems further support extensibility of ADLs:

LRQ4.1 External Data Type Integration: Many ADLs provide a fixed set of data types or allow to add user-defined data types described in an integrated, proprietary type definition language. This has the drawback, that all communication data types have to be realized twice: once for the architectural model and once for the concrete implementation. This redundant task can be avoided if an ADL is able to integrate externally defined data type definitions into its type system. Nevertheless, an abstract data type system still has to enable the development of type checks as well

as language processing tools which have to be independent from a concrete type definition language.

LRQ4.2 **Available Default Data Type System**: A default data type system is necessary for an ADL which is ready to use. Thus, a default implementation of the external data type system is needed. Suitable type definition languages for this purpose are, e.g., the target implementation language of the modeled system or an abstract modeling language which can also be used to generate concrete data type definitions.

LRQ5 **Libraries**: Another impediment for the application of ADLs in practice is missing or complicated reusability of architectural models [Woo05]. To enable controlled reuse, modular models which can be packaged into libraries are needed. McVeigh et al. [MKM06] define requirements for component reuse that have to be fulfilled by a library concept for ADLs. These are:

Alter: Alteration of library components to adapt them for reuse in the context of another system has to be possible.

NoImpact: Alterations of a component must neither impact other users of the component nor the provider of the library.

NoSource: Reuse has to work even if the source code is not available.

Upgrade: It has to be possible for users of a library component to accept an update of the component. Even if the updated component has been locally altered for reuse.

While the requirements **Alter** and **NoImpact** mostly impose requirements on the underlying ADL, the other requirements affect the technical realization of a component library. Thus, they are integrated into the following subrequirements:

LRQ5.1 **Version Control:** To unambigously indentify a library, it has to have a unique name and a version number. In this way, library users are able to explicitly choose which version of a library is used.

LRQ5.2 **Transparent:** Adapted from the library concept of Java, it has to be transparent for the modeler whether a subcomponent instantiates a component provided by a library or a locally defined component. Thus, the integration of library components has to be realized on a transparent technical layer.

LRQ5.3 **Intellectual Property:** It has to be possible to release closed source libraries without the source code of the models and component implementations. Further, it has to be irrelevant to the library user whether the library provides its sources or not. By applying object oriented encapsulation properties to model libraries, the intellectual property of the contained components and implementations can be retained.

2.2 Simulation Requirements

Simulations of architectural models can be used to explore and validate models of interactive systems in an agile manner. Thus, they are suitable to validate models in an early development stage (see *RQ4*). To simplify the implementation of atomic components, architectural modeling can be combined with architectural programming. According to Baumeister et al. [BHH+06], APLs enforce the properties of architectural descriptions, such as strong encapsulation and type

safe communication, on the programming level. This is achieved by integrating architectural modeling languages into GPLs. A loose coupling to the concrete behavior implementations avoids restricting such an ADL to a single target GPL. Thus, an agile workbench is necessary in which architectural models are the main development artifacts and behavior implementations can be flexibly integrated into a derived simulation. In the following, simulation specific requirements are listed which determine important properties of architecture simulations.

SRQ1 Platform Independence: In distributed development teams, heterogeneous computing platforms can be in use. This comprises distinct processor architectures as well as operating systems. To support distributed development of architectural models, it is necessary that architectural simulations are executable on various computing platforms. Especially, if component libraries (see requirement *LRQ5*) contain generated or handwritten simulation specific implementations, it is very expensive to compile these implementations for every used computing platform. Thus, a computing platform independent implementation language is most suitable to implement architectural simulations.

SRQ2 External Component Implementation: To avoid a binding of an extendable ADL to a concrete target GPL, it is necessary to define the implementation of atomic components in external artifacts. Hence, mechanisms are needed to transparently integrate these external behavior implementations into the simulation. To support validation of components (see requirement *SRQ7*), the component tester has to be able to interchange implementations before execution-time, e.g., with mock implementations.

SRQ3 Mathematical Foundation: To allow for early validations of components a simulation is necessary which is based on a mathematical framework. For the architectural style of interactive systems, the simulation has to support asynchronous, timed, and parallel event processing. Simulated timed streams with explicit messages allow to simulate communication as well as time progress. These streams model the communication history between connected components (see [BS01, Chapter 4]).

SRQ4 Component Timing Classification: To allow simulation of systems which contain combinations of components with different timings, it is necessary to support all timing domains of the architectural style (see requirement *LRQ1.1.3*).

SRQ5 Simulation Time: To decouple an architectural simulation from real time, it is necessary to explicitly simulate time (simulation time). In this way, timed behavior of components of an interactive system can be validated and the simulation can be executed much faster than real time.

SRQ6 Distribution: By distributing the simulation of components of a complex, computation intense system to distinct physical nodes, the overall execution time of the simulation is reduced. Nevertheless, distribution also introduces additional overhead caused by message transmission between the physical nodes. To reduce this overhead, synchronization of the simulation time between the distinct physical nodes should not cause any additional overhead compared to a non-distributed simulation.

SRQ7 Component Testing: To validate components and architectures in early development stages, they have to be testable. Automatic test execution further supports distributed and agile component development, since all developers are able to execute tests defined by other developers. The following properties have to hold.

SRQ7.1 Determinism: A deterministic simulation is necessary to achieve repeatable tests. Thus, repeatedly executed system-simulations with the same input always produce the same simulation results. This, however, does not hold if non-deterministic components are involved in the simulated system.

SRQ7.2 Black-box Behavior: To verify whether the behavior of an atomic or decomposed component corresponds to its requirements, component black-box tests are necessary. Black-box tests compare actual with expected reactions of components that are stimulated with a given input.

SRQ7.3 White-box Behavior: To verify the interaction of subcomponents in a decomposed architecture, e.g., to check, whether expected signal flows are present within a component, it is necessary to support white-box tests.

SRQ7.4 Mocks: For testing decomposed architectures it is necessary to substitute subcomponents by mocks. In this way, irrelevant or non-deterministic subcomponents can be replaced.

SRQ7.5 Timed Behavior: To validate, whether certain non-functional timing related requirements are held by a component or a composition of components, the timing of messages has to be explicitly testable.

SRQ8 Extensibility: To adjust the simulation to an extended variant of the ADL (see requirement *LRQ3*), the simulation infrastructure has to be extendable, too.

SRQ9 Scheduler: To decouple simulation time from real time (see requirement *SRQ5*) and to assure deterministic simulation results (see requirement *SRQ7.1*), explicit simulation scheduling is necessary. Further, the following subrequirements allow to adjust the simulation with custom schedulers.

SRQ9.1 Default: To simulate stream-based message passing as well as the progress of time, a default scheduler is necessary. In this way, generated simulations are ready to use.

SRQ9.2 Customized Schedulers: Customized schedulers in the simulation are necessary to integrate new scheduling strategies for extended ADL variants into the simulation.

SRQ9.3 Multiple Active Schedulers: Every scheduled component is controlled solely by a single scheduler. However, by using multiple schedulers in the simulation of a decomposed architecture, it is possible to optimize schedulers to specific component types.

SRQ10 Optimizations: By optimizing runtime and memory use, a fast execution of the simulation is possible. Optimization strategies can be realized by the simulation runtime infrastructure, the generator, and the default scheduler.

2.3 Currently existing ADLs

In the last decades a broad set of ADLs and corresponding tools have been developed. These can be grouped into first and second generation ADLs [AS10]. The former mostly focus on a specific architectural style or have a certain unique feature [MT00, DvdHT02]. A selected set of first generation ADLs has been compared by Medvidovic and Tayler with a well defined classification and comparison framework in [MT00]. Among these are:

- Aesop [GAO94, Gar95] which allows to define architectural styles. An Aesop style enforces certain architectural rules and properties similar to context conditions, to support software architects in modeling specific architectures.
- Darwin [MDEK95, MK96] aims at modeling distributed, dynamic systems. It is based on an underlying operation model that is described in the π-calculus.
- MetaH [BEJV93] focuses on the interaction between software and hardware in real-time systems.
- Rapide [LKA+95, LV95] is designed to model event-based architectures. It provides tools that simulate the interaction between modeled components using partially ordered event sets.
- Wright [All97, AG97] introduces explicit connectors as first level modeling elements. Connector specifications can be designed and checked with a variant of the CSP [Hoa85].

Since these languages do not focus on extensibility (see requirement *LRQ3*) and mostly cannot be used as an APL (see Section 2.2), they can be neglected in the following. In contrast, second generation ADLs took over fundamental architectural concepts common to first generation ADLs and further add concepts that allow architectural interchange and extensions [AS10]. Among these are:

- the AADL [FGH06, FG12, SAE12],
- Acme [GMW97, www14t],
- ADML [www14c],
- xADL [DvdHT01, DvdHT02, DvdHT05, www14x], as well as
- UML [OMG11b] and SysML [OMG12].

Since MontiArc also aims at architectural programming, it has to be compared with languages that can be classified as APLs, too. APLs are GPLs with integrated architectural elements as primitive language constructs [BHH+06]. They lift the advantages of ADLs to the programming level to enforce strong encapsulation and type safe communication. Nevertheless, in a more abstract view, also ADLs that provide target code generation or (interactive) simulations, that allow to rapidly explore the behavior of the modeled architecture, can be regarded as an APL. Thus, the compared APLs are:

- ArchJava [ACN02b, ACN02a] and JAVA/A [BHH+06],
- AutoFocus [BHS99, HF10, HST10], and
- Ptolemy II [EJL+03, Pto14, www11].

In the following sections, the listed second generation ADLs and APLs are compared to MontiArc considering the requirements given in Section 2.1 and 2.2.

2.3.1 AADL

The Architecture Analysis and Design Language (AADL) [FGH06, FG12, SAE12] is an ADL standardized by the Society for Automotive Engineers. It focuses on the early analysis of timing and scheduling properties of static hardware and software architectures of real-time embedded systems, e.g., from the automotive, avionic, robotics, and aerospace domain [HWF+10, www14b]. According to an industrial survey performed by Malavolta et al. [MLM+13], the AADL is one of the top-used ADLs in industry. It is designed as an extendable ADL. There-

fore, it provides the annex concept, which can be used to refine and adapt the given architectural style. Beside its application in industry, AADL is also used and extended in many industrial and scientific projects. For instance, the COMPASS Project [BCK+09, www14g], which focuses on specification and analysis techniques, developed formal methods to enable safety, dependability, and performance analyses of AADL models [BCK+14, NBKN14]. AADS [VGV09] provides a simulation of AADL models by transforming them to SystemC [IEE11]. It supports performance analyses of modeled architectures and the validation of timing constraints. The open-source toolchain TASTE [www14s] combines AADL with other languages, such as Modelica [www14l], SysML [OMG12], or Matlab/Simulink [www13d], to a toolchain for embedded software development. The stand-alone AADL model processors Ocarina [www14m] provides target code generators for ADA and C as well as analyses tools.

AADL models are mostly defined textual. A compilation unit (i.e., a file) represents the definition of a package with public and private areas. The Graphical AADL Notation Annex defines a standardized graphical notation and the XML/XMI Interchange Format Annex defines a XML format which is used to interact with other tools. Both annexes are standardized in [SAE14]. The following analysis is based on [FGH06, FG12, www14b] and the open-source AADL tool Osate2 [www14o].

Architectural Style: The AADL modeling language provides a complex architectural style which focuses on the development of embedded real-time systems. Modeled components can either represent application software, the execution platform, or a composition of components. AADL explicitly distinguishes between component interfaces and component implementations. The former define a set of features as interaction points with the environment. The latter implement a component interface and define the architectural configuration of a component.

A rich set of predefined properties, such as memory use or execution time, allow to model the execution behavior of soft- and hardware components in detail. Advanced modeling concepts like modes or message flows are provided that allow to model different operation modes of components or to analyze communication properties of the modeled system. AADL also provides four different connection kinds: port, parameter, and component access connections as well as subprogram calls. Port connections, which represent delayed or immediate transmission, are further grouped into unqueued data connections, queued event connections, and event data connections. Compared to the FOCUS architectural style, immediate event data connections correspond to FOCUS channels (see requirement *LRQ1.3*). Parameter connections model data for accessing subprograms, component access connections model direct access to contained subcomponents, e.g., access to data or a bus. Finally, subprogram calls model the call sequence of contained subprogram components.

A timing classification concept (see requirement *LRQ1.1.3*) does not exist in AADL. Nevertheless, the component execution semantics is based on a thread execution model [FGH06]. By analyzing the used port kinds (delayed or undelayed) and properties, such as the execution frequency of components, over- and undersampling components can be identified and execution schedules can be calculated [FGH06]. Also structural or behavioral constraints over AADL models can be defined using the Constraint Annex [HG13]. The former are used to define project specific rules for AADL properties, the latter are used to define assume/guarantee style specifications for components and systems.

Overall, the AADL architectural style allows to model embedded real-time systems in great detail, which comprises the interaction between software and hardware, implementation details of software components, and concrete software deployment. Since this thesis focuses on the logical interaction of interactive systems (see requirement *LRQ1*), most of these detailed model elements are not necessary for this purpose. Especially, concepts, such as component access connections, which reveal internal implementation details, and data ports, that model shared data (see requirement *LRQ1.1.5*), are not carried over.

Usability: By default, Osate2 does not provide a command line interface (CLI) that allows to automate analyses or code generation tasks. Nevertheless, several other tools are able to process AADL models with provided CLIs for batch processing. Among these are commercial tools, such as the AADL Inspector [www14a], focusing on the analysis of AADL models, or open source tools, like Ocarina [www14m] or RAMESES [www14p], providing several analyses, transformations, and C code generation. Since many AADL components and implementations are defined in a single model file, modular processing of AADL components is not possible.

The Osate2 IDE provides rudimentary active specification such as keyword proposals or an integrated validation of context conditions. The latter, which mostly focuses on the composition of categories and the validation of connections, are automatically checked in Osate2. Also approaches like the annex language REAL [GH10], which is integrated in Ocarina, allow to define implied semantics of properties or patterns. Thus, it can be used to define user-specific adaptions of the AADL architectural style. However, since AADL models are usually pretty large, a more sophisticated active specification which allows to create valid connections or helps to define flows, would be helpful. Neither contained comments are used to further support the modeler nor is the modeler automatically informed about unconnected ports or subcomponents. To support distributed development, Osate2 can be extended with a revision control system (RCS) client since it is based on the Eclipse platform. Automatic validations of merged models are not available. Due to the large AADL community, many tutorials in the Internet help to model with AADL. These range from small getting started tutorials to tutorials that detail certain aspects of modeling.

Reusability and Extensibility: AADL is designed as an extendable language. On the one hand, it inherently supports the extension with new property sets. On the other hand, its concrete syntax can be extended with annexes [FGH06]. As already mentioned, several annexes exist in practice, some of them are even standardized by the SAE [SAE11, SAE14]. Osate2 implements the AADL language with Xtext [www13e]. Consequently, language elements can be added in a sublanguage that, e.g., defines an annex. Nevertheless, according to Völkel [Völ11, Chapter 9] this only affects the concrete and abstract syntax of the language. Infrastructure for compositional tool development is not provided. Also, developing a language extension for AADL requires the complete Osate2 source code [www14o] which leads to a heavy weight extension process.

The AADL itself does not provide an explicit extension point for behavioral definitions. Nevertheless, behavioral languages can be embedded into an annex. For example the standardized Behavior Annex [SAE11] provides a state-machine notation which can be used to implement the behavior of components. Several approaches [YHMP09, ÖBM10] define a formal semantics for the Behavior Annex with transformations to formally defined languages such as Timed Abstract

State Machines [OL07a, OL07b] or Real-Time Maude [ÖM07]. The transformed models are most often used for simulation or model checking (see below).

Type System: AADL provides a proprietary data type system integrated into the AADL language. This data type system allows to model custom data types using data components. Complex data types can be defined by aggregating data components in data component implementations. Since Osate2 is realized with Xtext, no support for language aggregation is available (see [Völ11]). Thus, external data type languages cannot be integrated.

Libraries: Osate2 is delivered with a set of libraries such as basic data types or default property sets. The contained models can be reused by importing a library into a package and referencing the models with their fully qualified name. Consequently, transparent reuse is only partially possible. Since the source code of a model has to be available for processing, it is not possible to release libraries in a binary form to protect the intellectual property of the models. Consequently, controlled reuse according to [MKM06] is also not possible.

Simulation: Several approaches and tools exist which allow to simulate AADL models. Most of these approaches translate AADL models into execution models which can then be explored in the corresponding tooling. Among these are:

- AADS [VGV09] and its successor AADS+ [VGV10]. Both translate AADL into a SCoPE [www14r] model in order to validate if AADL constraints are fulfilled for concrete deployments. SCoPE is a C++ library that extends SystemC [IEE11]. It allows to simulate C/C++ software code in hardware models described in SystemC. AADS initially focuses on the validation of global performance constraints and timing constraints of the contained components. AADS+ further translates abstract state machines defined in the AADL Behavior Annex [SAE11] to take concrete component behavior definitions into account.
- AADLtoTASM [BGL09]. An Osate2 plug-in that translates AADL models into equivalent Timed Abstract State Machines (TASMs) [OL07a, OL07b] which can then be simulated using the TASM Toolset. This translation allows to analyze timed properties of AADL models in a timed simulation of the equivalent TASM. A TASM specification is a set of machines, sub machines, and function machines which interact using shared variables. AADLtoTASM is based on an extended version of the AADL Behavior Annex which allows to annotate transitions with timing intervals that model the duration of the transition.
- Chkouri et al. [CRBS09] present an approach that translates AADL models into Behavior Interaction Priority (BIP) [BBS06] models. In this way, BIP's formal operational semantics based on labelled transition systems, is assigned to AADL. Using provided BIP tools allows to validate the transformed models and to ensure that properties, such as state invariants, deadlock-freedom, and schedulability, are met.
- Ocarina [www14m] is a stand-alone AADL model processor which provides code generators for C and Ada targets. This allows to explore the modeled system in the target language. The Ocarina tool-suite also contains mappings to timed and colored petri nets. In this way, model checking tools, such as TINA [www14v] or CPN-AMI [www14h], can be used to evaluate properties such as invariants, deadlock-freedom, liveness, and timed relations between events. Also the state space of the transformed models can be explored.

Another approach is elaborated by ADeS [TSS08]. This simulator aims at supporting all behavior information provided in AADL models. It is available as an Osate2 plugin, and, in

contrast to the previous approaches, interprets an AADL model and does not transform it to another execution model. It simulates active component categories such as threads, passive component categories like data components, and communication specific components such as bus components. It also considers modes and a simplified version of the Behavior Annex. ADeS does not support the integration of external component implementations natively, but in [TSS08] it is mentioned, that the simulation can be extended to also consider further AADL extension. However, a documentation of the needed extension points is not provided. Since the simulator exports simulation traces in an open XML format, custom simulation analysis tools can be developed more easily. The centralized trace generation, however, has the drawback that complex simulations cannot be distributed to distinct physical nodes.

The ADeS simulator is developed in Java as an Eclipse plugin. It is based on the infrastructure (parser, abstract syntax tree (AST), etc.) provided by Osate2. Thus, it can be executed by any computation platform which is supported by the Osate2 IDE. It implements an event-based simulation approach which completely decouples the simulation from real-time. The simulation is controlled by an integrated scheduler which cannot be exchanged easily with a custom scheduler. ADeS is tailored to an interactive analysis of the produced simulation traces and not for automatically executed component tests. Thus, exchanging components of a simulated system with mocks is not possible.

2.3.2 Acme and xADL

Acme [GMW97, www14t] and xADL [DvdHT01, DvdHT02, DvdHT05, www14x] are both designed to be extendable. Both offer a set of basic architectural modeling elements which are not tailored, but can be extended and adapted, to a certain system domain. To achieve this goal, both ADLs follow a different approach.

Acme has been initially designed to define an architectural interchange format which allows to translate architectural models from one ADL to another. It is a basic textual ADL defined with a BNF grammar and comes with an IDE named AcmeStudio [www14u] that provides a graphical front end for architectural modeling. Architectural styles are used to tailor the basic ADL to a certain domain. Such a style is defined by a family which declares new types such as components, connectors, ports, roles, and other elements, allowing to adjust Acme to a certain domain.

Two different approaches extend Acme with XML representations. The Architecture Description Markup Language (ADML) [www14c] defines the structure of the representation with a Document Type Definition (DTD). It can be extended to new languages, but the extensions cannot be used in existing tools [DvdHT01]. In contrast, xAcme [Sch01] defines the language using XML schemas as an extension of xArch [www14y], which itself already provides an XML syntax for basic architectural elements. In this way, existing xArch tooling can be reused for xAcme models.

For the same reason, xADL has been developed as an extension of xArch [DvdHT01]. Beside an improved extension mechanism (see below), it adds means to model architectural variants and versions. Further, it adds a generic placeholder for implementations of components and connectors. The IDE ArchStudio [www14e] integrates several xADL tools.

Architectural Style: Both languages, Acme and xADL, do not provide a predefined architectural style. However, they provide integrated mechanisms to define custom architectural styles. Both miss a concept to instantiate component definitions in another component. Consequently, each used component corresponds to a component definition that cannot be reused easily in another context. On that account, advanced concepts which improve reusability, e.g., configuration parameters, generic types, or inheritance, are not available. Acme defines component interfaces using directed ports that instantiate a port type from the current family. In xADL, interfaces only define an interaction point with other components. It is neither directed, nor typed. Both languages and the corresponding tools provide means to statically analyze models similar to context conditions. Acme (AcmeStudio) allows to define and attach invariants to model elements that are evaluated to check certain properties of the model. In this way, it is possible to tailor a family to a certain domain, e.g., by restricting which port types can be connected or which relations are allowed between elements of the family. In xADL (ArchStudio), a set of so called critics is provided for this purpose. It can be extended by installing new analysis tools. A list of default critics is given in [DvdHT05].

Usability: Both IDEs do not provide build infrastructure to automatically process models without user interaction and are delivered without an RCS client. However, since both tools are Eclipse-based, an RCS client can be installed manually. Active specification is not available in both IDEs. This further complicates modeling, since a huge number of options are available in the extendable IDEs. Distributed development of Acme models is simplified by the underlying textual syntax. In contrast, the XML syntax of xADL complicates distributed modeling [BLF14]. Both languages are introduced by tutorials. Unfortunately, these do not cover the rich set of available tool and modeling possibilities.

Reusability and Extensibility: Acme allows to define new families that add new component, connector, port, role, and element types to the language. A new type can extend an existing type and inherit its properties. However, this extension mechanism is restricted to the addition of properties and invariants. The set of core elements cannot be extended. Due to this restriction, it is not possible to easily integrate arbitrary behavior definition languages into Acme. Nevertheless, the described extension mechanism is directly integrated into AcmeStudio, which allows to immediately use the extended language.

In contrast, xADL allows to add and refine new language elements freely by defining a new XML Schema that extends the xADL Schema. It even provides an abstract implementation placeholder for component and connector implementations. It can be used to extend the language with concrete behavior definition languages. By default, Java can be used to add implementation details. Naslavsky et al. [NXD+04] present an extension for statecharts. Although it is possible to use extended languages in ArchStudio, some integration effort has to be carried out and existing tools have to be adjusted (see [www14x]).

Type System: Both languages do not offer a data type system that allows to restrict the communication type of ports. It should be possible to extend xADL with typed ports and analyses that ensure type-safe connections. However, a flexible integration of different data type systems is not possible due to the missing support for language aggregation.

Libraries: Instantiation of components is not supported in both languages (see above). Consequently, a library mechanism is not provided.

Simulation: A simulation of xADL or an extension is not available. Aldrich et al. [AGST04, AAAG+05] demonstrate a combination of Acme and ArchJava. The presented approach allows to synchronize Acme models with ArchJava components and thus execute Acme models as ArchJava components. More details of the ArchJava component execution are given in Section 2.3.4.

2.3.3 AutoFocus 3

AutoFocus 3 (AF3) [HST10, HF10, www14f] is a model-based development tool for embedded, reactive systems that aims at supporting all phases of the development process. It has been initially developed at the TU Munich [BHS99, Wil06] in cooperation with the spinoff Validas AG[www14w] and is now developed by the spinoff fortiss[www14i]. The graphical AF3 ADL mainly targets static architectures, but it is capable to model dynamic aspects with modes. AF3 provides a rich set of features to support consistent model-based development.

- Requirements and use cases can be defined and linked to architectural modeling elements with traces.
- Integrated checklists support the requirement definition process.
- The completeness of defined requirements and use cases can be automatically reviewed.
- Logical system architectures can be specified with a FOCUS [BDD+93, BS01] based ADL.
- The embedded target platform and the deployment can be specified. Based on this specification, the target schedule can be generated.
- An integrated FOCUS simulator [HST10] allows interactively explore the specified system.
- Target code generators are provided for Java and C.
- The model checker NuSMV [www15d] and the theorem prover Z3 [www15e] are integrated with AF3.
- Non-determinism analyses can be executed on system specifications.

Architectural Style: The architectural style of AF3 ADL is based on the FOCUS modeling framework [BS01] and implements the time-synchronous frame [HST10]. AF3 components are defined within *Component Architecture Roots* that correspond to a system component. Interfaces are defined by a set of directed and typed ports. Decomposition is achieved by defining inner components and their connection. Consequently, component instantiation is only partially supported (see libraries). The behavior of atomic components can be defined using automata. Since all elements of an AF3 project are stored in a single XML file, the concept of a compilation unit does not exist. It is not possible to define generic or configurable components and inheritance is not supported.

AF3 is based on a synchronous component execution semantics. Since AF3 components process time-synchronous streams and are either strongly or weakly causal [HST10], components can be executed with a static sequential schedule. An active component synchronously reads all its input values and writes all its output values in the current (weakly causal) or next (strongly causal) step. Furthermore, the behavior of components can be formally constrained with assumptions and guarantees, contracts, as well as patterns. Static analyses are available that, e.g.,

validate created connections or identify weakly causal feedback cycles. Documentation of components is achieved by linking requirements to a component and by adding comments within the property view of a component. These comments are shown when hovering over the component. Nevertheless, comments from a library component (see below) are not transferred to an instance of the library component.

Usability: The introduction to AF3 an the corresponding Eclipse-based rich client IDE is supported by a set of tutorials, picture books (e.g., [HLP+14]), and screen casts. AF3 does not provide a build infrastructure that enables model processing in an automated way. Executing test suits or performing model analyses rather depends on user interaction within the IDE. Furthermore, active specification is not supported by AF3, which complicates the definition of well-formed models. However, the aforementioned static analyses are automatically executed to inform the modeler about mistakes. Distributed development of AF3 models is not supported since a) no RCS client is integrated into AF3, b) the used XML data structure cannot be easily merged [BLF14], and c) automatically executed quality checks are not available.

Reusability and Extensibility: The AF3 ADL is not designed to be freely extendable. Consequently, it is not easily possible to add or refine language elements or add new behavior definition languages. At least no public documentation, methods, or tutorials are available. However, the AF3 architecture is designed in a modular way, divided into the following projects: architectural modeling, code behavior specifications, simulator, and code generators. Consequently, these modules should be reusable in another context.

Type System: AF3 provides an internal data type system that is restricted to the basic types int, boolean, double, and custom type definitions. The latter can be designed in a *Data Dictionary* that allows to model enumerations, structs, and functions. However, these are restricted to a combination of the mentioned basic types. It is not possible to integrate external data type systems into AF3.

Libraries: AF3 provides a library concept which allows to import components defined within the current model into the library and export them to other projects. Components from a library can be instantiated in other components and can only be modified within the library view. This guarantees, that a library component does not differ in distinct architectures or projects. Nevertheless, this concept has room for improvement. First, libraries can contain components with identical names. Second, libraries do not have a dedicated version number. Third, the sources of a library always have to be available. Consequently, according to McVeigh et al. [MKM06], reusing AF3 components is only possible in a restricted way.

Simulation AF3 contains a simulator which allows to perform an interactive, step-wise simulation of modeled systems. All input values of a system can be manually set, the values transmitted over channels can be displayed on all levels of the architecture, and active states and transitions of behavior defining state machines are highlighted. In this way, the interaction between contained components as well as modeled behavior can be interactively explored.

AF3 is developed as an Eclipse based rich client in Java for the most common operating systems Windows, Linux, and Mac. 32- and 64-Bit versions are available on the AF3 website [www14f]. Since the simulation is executed within the graphical AutoFOCUS client, it is restricted to these computing platforms with a connected screen device. It is not possible to distribute an AF3 simulation to distinct physical nodes.

The implementation of atomic AutoFOCUS components is given by state machines or a tabular behavior specification [HF10]. The latter can be used to implement stateless components with input/output relations. Beyond, it is not possible to attach an external implementation in the GPL of the target system to AutoFOCUS components. Alternative behavior definitions can be activated using modes. Nevertheless, the implementation of atomic and decomposed components cannot be freely exchanged before test execution.

The AF3 simulation is based on the time-synchronous FOCUS frame. Thus, it is possible to simulate asynchronous, timed, and parallel event processing. AF3 provides two timing domains for components: the weakly and the strongly causal time-synchronous domain. Components, which are able to process multiple messages within a time interval and components which are not aware of time cannot be modeled. AF3's simulation time is decoupled from real time. The simulation time is given by a global clock. Due to the time-synchronous foundation, this clock corresponds to the current simulation step.

Beside the interactive execution of a simulation, it is also possible to define tests that stimulate the modeled system or a single component to compare the actual with expected output. Tests are defined as a table which contains the stimuli as well as expected values for each time step of the testee [BMR12]. All tests of an AF3 project can be automatically executed within the AF3 client and produce deterministic test results. Thus, the timed black-box behavior of atomic and decomposed components and the complete system can be automatically tested. White-box tests that validate certain signal flows within a system cannot be defined.

AF3 provides a built-in simulation scheduler that executes the simulation. It analyzes the model and computes a static schedule in which the components are successively executed for each simulation step. It cannot be replaced with a custom scheduler to integrate scheduling strategies of the target system. Since the AF3 ADL does not focus on extensibility, the simulation cannot be extended as well.

2.3.4 ArchJava and Java/A

ArchJava [ACN02b, ACN02a] and JAVA/A [BHH+06] are APLs which both integrate architectural model elements as primitive language constructs into Java. Their main goal is to counter architectural erosion which is caused by software evolution [PW92]. Since architecture and implementation are combined in the same artifacts, both evolve jointly and erosion is prevented. Both languages allow to change the architecture of a system at runtime. In contrast to ArchJava, Java/A supports the definition of protocol state machines that can be model checked to validate whether components are deadlock-free.

Architectural Style: ArchJava and Java/A aim at defining local systems which communicate in a synchronous way. They define components as an instantiable type that can itself be composed to subcomponents. The component interface is defined by a set of required and provided ports that are internally connected to implementation methods. Consequently, a connector in both languages corresponds to a method call. Since parameters can be added to component constructors, configurable components can be implemented. However, generic components are not available in both languages. Only ArchJava supports component inheritance. Context conditions are defined by a formal semantics for Java/A [BHH+06] and a set of semantic rules for ArchJava [AJ002]. However, in Java/A it is possible to define syntactically correct models that

are compiled to invalid Java code. For example, missing types and duplicate variables are not checked. In ArchJava these flaws are detected before the Java code is generated.

Usability: Both languages provide a CLI that allows to compile the developed programs to regular Java code. ArchJava has been integrated into the AcmeStudio [AGST04] once. However, this integration is outdated and does not work with the most recent versions of AcmeStudio and ArchJava. Consequently, ArchJava and Java/A programs have to be developed in a plain text editor, which is an error-prone and cumbersome task. Beyond, no tutorials are available that guide the user through the languages. Nevertheless, both CLIs can be integrated in automatic builds and the textual languages can easily be controlled by an RCS.

Reusability and Extensibility: Both languages are not designed to be extended.

Type System: Both languages use the Java type system for data types which cannot be replaced with an external type system.

Libraries: Theoretically, it should be possible to bundle components together with the generated and compiled Java code in a jar file and add it to the Java classpath. However, tests have shown that ArchJava and Java/A cannot handle references to components within a jar file.

Simulation: Both languages provide a compiler which translates components into plain Java code that depends on classes of a static runtime environment (RTE). In this way, ArchJava and Java/A aim at target code generation and not at simulating the modeled system. Since components from both architectural styles synchronously interact with direct method calls and no scheduler is available to decouple components, system execution is coupled to real time. Timing domains that influence timed component behavior cannot be selected.

Both compilers produce plain Java code which solely depends on the corresponding RTE. Thus, the generated code can be executed on any computation platform with an available Java Virtual Machine (JVM). It can be tested like any regular Java program, e.g., with JUnit [www13c]. In both modeling languages, subcomponents are directly instantiated with a `new` statement which is directly reflected in the generated code. In the generated code of ArchJava, subcomponents are represented by a static final field. Thus, they cannot be replaced with mocks for testing. In contrast, subcomponents in the generated code of Java/A are managed by a component manager which allows to reconfigure components at runtime. In this way, subcomponents can be replaced with mocks for testing. This mechanism can also be used to replace the generated implementation of a component with an external one.

ArchJava and Java/A components are usually restricted to a single execution thread and cannot be distributed easily. Nevertheless, components can be developed that provide means for distributed communication, e.g., using TCP ports. This communication, however, is not type-checked anymore and thus can break architectural integrity of the developed system.

The RTEs of both languages can be extended using class inheritance. To use the extended RTEs, the compilers need to be adjusted. However, these are not documented and do not provide suitable extension points for this purpose. Theoretically, an extended ArchJava compiler can be derived from the available ArchJava source code. The sources of the Java/A compiler are not available.

2.3.5 Ptolemy II

Ptolemy II [EJL+03, Pto14, www11] is a framework developed by the EECS Department of the University of California that supports graphical modeling, interactive simulation, and design of concurrent, real-time, embedded systems. The runtime semantics of a model is defined by so called directors which implement a model of computation. The provided *Discrete Event* director is most suitable to model distributed, event-processing systems. One key feature of Ptolemy II is the hierarchical combination and exploration of different directors, e.g., to model cyber-physical systems [Lee10].

Architectural Style: Ptolemy II aims at modeling concurrent, real-time, embedded systems. It focuses on the mixture of mechanic, electric, electronic, and software components. A Ptolemy II model corresponds to a decomposed component. Its configuration is given by decomposition to interacting actors. Composite actors can be further decomposed with actors instantiated from predefined libraries (sinks, sources, etc.). The execution semantic of an actor is controlled with directors. The behavior of actors can be defined with *ModalModels* that contain state machines or by implementing the behavior, e.g., in Java or Python. The interface of an actor can be defined by directed ports. Nevertheless, the data type of a port is not defined explicitly. It is rather automatically determined when connecting it to another port. On that account, most of the consistency rules are not checked during modeling time. They are not checked until the model is executed, which often leads to invalid models. Since ports are nearly untyped, generic actors are not needed. A model corresponds to an XML file that stores the contained actors. Further, actors can be stored in a user library that is realized as a regular model, too. Ptolemy II supports actor instantiation and inheritance. Unfortunately, both relations are not visible to the modeler in the concrete syntax. Furthermore, configuration parameters as well as constraints can be defined for actors and a documentation can be attached.

Usability: Ptolemy II provides a graphical user interface named Vergil that can be regarded as an IDE. The introduction to Ptolemy II and Vergil is supported by tutorials (e.g., [Pto14, Chapter 2]) that introduce to the modeling and director concepts. Vergil does not provide any automatic build infrastructure for model processing and therefore depends on user interaction. Furthermore, distributed development of models is complicated due to a missing integrated RCS client and the underlying XML model format.

Reusability and Extensibility: Ptolemy II can be extended with new directors or custom actors. Beyond, it is not designed to extend the language itself. Consequently, it is not possible to add or refine language elements or add new behavior definition languages. Further, it is not possible to reuse single parts of Ptolemy II or Vergil, since it is not structured into independent modules.

Type System: Ptolemy II provides an internal data type system initially presented in [Xio02]. Data types are not explicitly declared, e.g., for ports, but inferred based on their usage [Pto14, Chapter 14]. The type system is checked just prior to the execution of a model and not during design-time. The type system itself can be extended [Xio02]. Nevertheless, it is not designed to adapt external type systems such as class diagrams (CDs).

Libraries: Vergil contains a library concept that allows to store user defined actors within a library to reuse them in different models. Instantiating such a library actor creates a copy of the actor in the current model. Consequently, the relation to the library actor is lost and changes

made in the library actor or in its instance do not affect the other one. Hence, it is possible to alter a library actor, but if the library actor is updated, there is no easy way to reintegrate these updates into the instance. Furthermore, libraries do not have a version number and at least the XML representation of a library has to be present in source form. Summing up, only restricted reuse according to [MKM06] is possible.

Simulation: Vergil provides a simulation engine which allows to interactively simulate modeled actors within their corresponding domains. Since Vergil is developed in Java, the modeling IDE as well as the simulation are platform independent and can be executed on various computing platforms.

The behavior of user defined actors can be implemented with modal models or externally, e.g., in Java or Python. Even though Vergil allows to integrate actors developed in Java, no concrete support for Java actor development is integrated. In fact, Java actors have to be developed in an external IDE, such as Eclipse, compiled, and manually added to Vergil's classpath [Pto14, Section 12.4], which is a quite cumbersome process. Furthermore, it is possible to implement actors in Java that break the type system. Since the type of a port is not fixed at design-time, an untyped token object is used to represent messages between actors. Consequently, it is possible to send messages with the wrong data type with a Java actor which leads to runtime exceptions.

In Ptolemy II simulations time is hierarchically controlled by the director of the top level hierarchy. A director on a lower hierarchy level obtains the current model from the director on the next higher level. Ptolemy II supports a model of superdense time [Pto14, Section 1.7] where a time value is a pair (t, n), a combination of the model time t and a microstep n. It is comparable to the time model of FOCUS, where t corresponds to the number of a time interval and n corresponds to the index of a message within a time interval. The simulation time is decoupled from real time with a default time resolution of one tenth of a nanosecond.

The provided directors offer a concept similar to component timing domains. Each director implements a suitable scheduling strategy for the corresponding domain. Since new domains can be integrated and domains can be hierarchically combined, customized schedulers can be introduced and multiple schedulers can be executed in a single simulation.

The discrete-event (DE) director simulates timed, asynchronous, event-based communication of (weakly) causal actors. The DE domain corresponds to the timed FOCUS domain. The simulation is executed with a global event queue which propagates events to actors according to their time stamp. In the homogeneous synchronous dataflow domain (SDF), actors consume a single token (event) on each input port and produce a token on each output port. Each actor is triggered once in a statically computed schedule. By default, the SDF domain does not use a notion of time. Nevertheless, it can be configured to advance the simulation time by a fix amount in each simulation step. This domain corresponds to the time-synchronous FOCUS domain in which each time interval contains a single data message. The process network (PN) domain models concurrent processes which communicate synchronously with message passing. Since this domain is untimed, it corresponds to the FOCUS domain of untimed streams.

Vergil does not directly support a distribution of the simulation to distinct physical nodes. While the process network domain is naturally suitable to be distributed, each actor is executed in a single thread, the other domains are not. The underlying event scheduling with a global event queue as well as complex interactions between directors of different domains hamper a

distributed simulation.

Ptolemy II focuses on the interactive exploration of modeled systems. Admittedly, the simulation is deterministic, but Vergil does not provide means for automated test executions. Further, the implementation of an actor cannot be mocked without changing the model.

2.3.6 UML and SysML

The Unified Modeling Language (UML) [OMG11b], which is standardized by the Object Management Group (OMG), is a collection of 14 different diagram types that allow to model the structure and behavior of software systems. Although it has some shortcomings for the purpose of system engineering [Hau06], it is considered as ADL in many works [Sel98, ICG+04, MDT07, MLM+13]. According to [MLM+13], UML is the mostly used set of languages for architectural modeling in industry. However, due to some limitations, it is rather used for design and documentation of architectures and not for automatic analyses or code generation.

To overcome these drawbacks of UML, the System Modeling Language (SysML) [OMG12] has been introduced by the OMG in 2006. It defines an UML 2.0 profile which consists of a subset of UML diagrams, adds new diagrams, and modifies existing diagrams to tailor UML for system engineering [OMG12]. However, due to its generic nature it still does not define a concrete architectural style for a specific domain.

Consequently, UML and SysML can be regarded as generic ADLs, whose constructs lack in formal semantics and therefore may become a source of ambiguity and inconsistency in some cases [Pan10]. Further, Petre [Pet13] reported that caused by concerns with model consistency, more than half of developers do not have a real interest in modeling technology. To overcome the issue of ambiguity, some approaches, such as the system model [BCR06, BCR07a, BCR07b], define a formal and extendable semantics for UML diagrams. However, it only covers a subset of the available diagrams such as CDs [CGR08a] or statecharts (SCs) [CGR08b].

Architectural Style: As already mentioned, UML and SysML have a rather generic architectural style. Nevertheless, several profiles exist that tailor UML or SysML to a specific domain. Kukkala et al. [KRH+05] provide an UML 2.0 profile for the design of embedded systems. Cechticky et al. [CEP+06] defined a profile that focuses on reusability of components. MARTE [OMG11a] is an UML profile for modeling and analysis of real-time embedded systems. A prominent UML profile named UML-RT is defined by Selic [Sel98]. It provides the concepts originally defined in the ROOM modeling language [SGW94] by introducing stereotypes and constraints to model protocols, protocol roles, ports, and components as a capsule that hides its internal implementation. It is, for example, supported by the proprietary IBM Rational Rose and the open source tool Papyrus [www15b].

Usability: A large set of different tools support modeling with UML and SysML. Enterprise Architect, Rational Rhapsody Developer, and MagicDraw are popular commercial tools. Papyrus [www15b] is an open source tool that is also considered for the following comparison. It is part of the PolarSys [www15c] platform, the successor of TOPCASED, which combines open source tools for the development of embedded systems. Most of these tools have a different focus. While Enterprise Architect focuses on constructive modeling and code generation, Papyrus concentrates on model synchronization aspects and the integration of user-defined profiles.

The majority of these tools are not suitable to automatically process models without user interaction [Pan10]. Since most of them persist produced models in the XMI format, a standardized XML format which is mostly used to interchange UML models, external tools can be used to automatically process the produced models. However, these external tools have to interpret custom profiles the same way as the used IDE, which is not always the case. Most often, context condition checks are directly integrated into the IDE. These are either executed automatically in the background or manually. Integrated Wizards and tutorials support the modeler while creating new modeling projects or models.

Due to the textual notation, models stored in XML can be managed in an RCS. Nevertheless, comparison of different versions requires extra effort with tools such as EMF compare, and merging of XML models is not recommended [AP11]. An alternative to textual revision control is provided by the CDO Model Repositories [www15a]. It offers persistence of and access to EMF models to support the collaboration of Eclipse-based modeling tools. The concrete model is managed on the server and accessed with atomic transactions. APIs are provided to compare and merge different branches of a model. Thus, it solves the aforementioned XML drawbacks by adding another tooling and complexity layer.

Reusability and Extendability: The UML provides an extension mechanism to define profiles that allow to specialize the generic forms of UML modeling elements to more application-specific variants [Hau06]. The intention of the profile mechanism is to adapt an existing meta-model by adding new constructs and not by modifying existing modeling elements [OMG11b]. It allows to add new syntax to the language, to define and refine the semantics of language elements, and to add constraints that restrict the way how language elements can be used. This light-weight extension mechanism is often used to tailor the UML for architectural modeling and system engineering [Hau06]. For example Kukkala et al. [KRH+05] defined a profile that aims at architectural modeling of embedded systems and Ivers et al. [ICG+04] describe how to use UML for the documentation of component and connector architectures. Most often, these profiles lead to non-standard UML. Standard and non-standard model kinds cannot reference each other. Since concrete implementation of extension mechanisms are tool dependent, the interchange of models between different UML tools is hindered [Hau06].

Type System: Flow ports of SysML block diagrams reference flow properties which specify the concrete item flow between ports. Thus, SysML flow ports reference model elements which are based on the same meta-model. Conceptually, this meta-model can be extended and a custom meta-class for flow properties can be defined as a subclass of the flow property meta-class. In contrast to the light-weight profiling extension mechanism, UML meta-model extensions are rather not supported by existing UML modeling tools. Consequently, UML and SysML modeling tools provide a default data type system based on the implemented meta-model which cannot be easily extended.

Libraries: UML and SysML support the instantiation of types by references to the type definition. Thus, the implementation of a library mechanism depends on the concretely used modeling tool. If a tool is able to resolve external models, a library concept which fulfills the requirements given by McVeigh et al. [MKM06] can be realized. E.g., Papyrus does not provide any means to use models of another project respectively library within the current project. It integrates the CDO repository infrastructure, but it is solely used for cooperative modeling of

a single project. Since the UML/P [Rum11, Rum12, Sch12] is developed based on the same technical foundation as MontiArc, i.e., MontiCore [Kra10], it also allows to reference models which have been packaged in jar files. In this way, reusable libraries can be developed.

Simulation: The aforementioned missing concrete architectural style of UML hampers the possibility to simulate UML models. Common UML models are rather used for the communication of the system design. Nevertheless, several approaches define a more concrete architectural style with an UML profile which assigns a concrete runtime-semantic and provides a simulation or target code generation. In the following, three different approaches are summarized: the UML/P, OMNeT++, and a SysML simulation.

The UML/P provides a method for the generative development of object oriented systems based on a reduced set of UML diagrams. Therefore, CDs are used to constructively define the structure of a system, SCs define the behavior of methods. A combination of object diagrams (ODs), sequence diagrams (SDs), and a test-specification language is used to specify tests in a model-based way. All these languages are combined with an action language that allows to directly implement complex actions within models. It further allows to easily integrate handwritten code into the generated system, which can be used to integrate external behavior implementations. The UML/P itself is developed in Java and the provided generators produce Java code. Thus, it is considered to be platform independent. Since the UML/P provides a constructive way for model-based system development, the generated code depends on a concrete RTE that is rather not suited for simulations. Thus, no scheduler is provided which decouples system execution from real-time and a timing cannot be selected. However, the language infrastructure presented in [Sch12] provides an extendable set of languages and an extendable language processing framework, which can be tailored to specific needs of a system-simulation.

Other approaches transform (profiled) UML models to more formal or executable languages, which allows to simulate the modeled system. For example Michael et al. [MSMB04] transform UML/RT models to OMNeT++ [www14n], a discrete event simulator, primarily designed for the simulation of network protocols within parallel, distributed systems, which is available since 1997 [VH08]. OMNeT++ is developed in C++. Thus, specific compilation is needed and existing simulation models for protocols have to be imported and compiled for the current computation platform. Compound modules are defined in the NED language, a domain specific language (DSL) for the definition of simulations. The implementation of simple modules (comparable to atomic components) is directly given in C++. Nevertheless, the JSimpleModule extension allows to implement simple modules in Java and the CSharpSimpleModule extension provides this feature for C#.

OMNeT++ is not based on a formally defined semantics and modules are not classified to distinct timings. The simulation time is represented by a global clock which decouples the simulation from real-time. Time-stamps are assigned to every event which is processed by the simulation. Şekercioğlu et al. [ŞVE03] present an approach to distribute compound OMNeT++ modules to distinct physical nodes. The modules are synchronized over named pipes or files. Since events are stored and organized in a future event set, synchronization of time needs some extra effort [Cra96].

The aim of OMNeT++ is to gather simulation results and analyse them afterwards in a graphical way [VH08]. Nevertheless, it provides the opp_test tool to define and execute simulations

and compare the output with predefined patterns [Var14, Chapter 15]. It is mostly suitable for black-box tests of modules. To implement white-box tests, new compound modules have to be created to manually replace some of the involved modules with mocks. Thus, this approach cannot be applied without changing the original compound module or a copy of it. Since time is an attribute of the exchanged events in OMNeT++, the timed behavior can be tested.

OMNeT++ provides a plug-in extension mechanism which allows to extend the existing simulation in a modular way [Var14, Chapter 17]. It allows to add new random number generators, schedulers, configuration providers, output and snapshot managers. A set of default schedulers is provided, e.g., a sequential scheduler, and a real-time scheduler, and can be selected before the simulation is started. Only a single scheduler can be active during the execution of a simulation.

SysML introduces parametric diagrams, a specialized form of internal block diagrams, which allow to model relations between properties of system parts [OMG12]. Constraints are modeled as a block containing an equation. Parameters are variables within constraints which are modeled as incoming ports. Value properties are quantifiable characteristics of system parts which are calculated as the result of a constraint. Thus, complex (physical) properties can be aggregated by combining constraint blocks and regular block. Parametric diagrams can be simulated by parametric solvers such as IBM Rational Melody, which is based on the approach presented in [PBF+07a, PBF+07b]. SysML SDs are used as a test language which interrelate requirement diagrams and blocks to simulate the latter. Constraints are calculated by external tools such as Mathematica or OpenModelica. The simulation of time is not yet supported. Kawahara et al. [KND+09] present another approach which focuses on the concrete combination of SysML with numerical solvers such as Matlab/Simulink or Modellica. Event-based interaction and communication between blocks is modeled in SysML, the implementation of continuous processes is given in Matlab/Simulink. In contrast to [PBF+07a], the time management of the Simulink simulation is explicitly controlled by a provided time management. In this way, SysML and Simulink are synchronized and the simulation time is decoupled from real-time. Nevertheless, both approaches rather focus on the simulation of physical properties than on the interaction between logical system parts.

2.3.7 Summary

The features of the related languages and corresponding simulation frameworks are summarized in Table 2.1 and Table 2.2. Please note, Acme and xADL are not considered in the latter table since no simulation environments exist for these languages. It is shown, that none of the compared ADLs and APLs fulfills all requirements given in Section 2.1 and 2.2 to the full extent. Most of them are capable to express the structure of interactive systems and are therefore suitable to communicate their design. Some lack in usability, some cannot be extended to define new architectural styles, and almost all do not provide a well defined library mechanism that simplifies reuse and distributed development. Also the non-interactive execution and validation of simulations, which further supports collaborative, distributed, and agile development of components, is rather not provided by the examined frameworks. In fact, they rely on user-interaction to execute or analyse the produced simulation results.

One practical way to fulfill the listed requirements could be to extend an existing ADL and the corresponding tools. Most suitable candidates which provide extensibility are AADL, Acme

and xADL. The AADL OSATE2 implementation is realized with XText. It admittedly allows to extend the syntax of a language but does not support the compositional development of extended tools [Völ11, Chapter 9]. Acme defines a common architectural interchange format with a BNF grammar. The provided tooling, however, is based on handwritten parsers which cannot be easily extended. Thus, the complete language processing tooling for an extension of Acme has to be developed from the scratch. xADL can be extended by defining new XML schemas that extend the xADL base schema. Even though extended xADL dialects can be used in the existing IDE ArchStudio, existing tools have to be adjusted for every dialect. Further, xADL models are stored in XML syntax, which on the one hand forces the use of a heavy-weight graphical modeling editor, and on the other hand hampers distributed development.

Feature	AADL	Acme	xADL	AutoFocus3	ArchJava & Java/A	Ptolemy II	UML & SysML
Architectural Style	embedded real-time systems	generic	generic	interactive systems	local, synchronous Java systems	concurrent embedded real-time systems, CPS	generic
Usability							
IDE	various, e.g., Osate2	AcmeStudio	ArchStudio	AF3	AcmeStudio / no	Vergil	various, e.g., Papyrus
Active Spec.	partial	no	no	no	no	no	partial
Tool Infrastructure	no, external	no, external CLI	no	no	CLI	no	no, external
Context Conditions	yes	extendable	extendable	yes	few	no	yes
Wizard	yes	yes	yes	yes	no	yes	yes
Distributed Dev.	partial	yes	partial	no	yes	no	partial, CDO MR[1]
Tutorial	yes	partial	partial	yes	no	yes	yes
Reusability and Extensibility	property sets & annex	family	XML schema ext.	no	no	no	profile, meta-model
Model Elements	annex or Xtext extension	refine existing, add properties, invariants	yes	no	no	no	add new
Behavior Ext. Point	annex	no	yes	no	no	no	no
Modularity	partial	yes	yes	partial	no	no	partial

Table 2.1 continued on next page

[1]CDO Model Repositories: `http://www.eclipse.org/cdo/documentation/`

Feature	AADL	Acme	xADL	AutoFocus3	ArchJava & Java/A	Ptolemy II	UML & SysML
Type System							
External Data TS	no	no	yes, n/a	no	no	no	meta-model extension, n/a
Default Data TS	proprietary	no	no	proprietary	Java	proprietary	proprietary, meta-model based
Libraries							
Version Control	t.d., manual[2]	no	no	no	no	no	t.d., no[3]
Transparent	t.d., partial	no	no	partial	no	no	t.d., no
Intellectual Property	t.d., no	no	no	no	no	no	t.d., no

Table 2.1: Overview of language features provided by the examined related work.

Feature	AADL	AutoFocus3	ArchJava & Java/A	Ptolemy II	UML & SysML		
Considered	ADeS [TSS08]	AF3 [HF10]	Compiler & RTE	Vergil [Pto14]	UML/P [Sch12]	OMNeT++ [MSMB04]	SysML [PBF+07a]
Platform Indep.	partial, OSATE plugin	mostly	Java	Java	Java	rather not: C++	no
Ext. Comp. Impl.	no	no	yes	yes	yes	yes	no

Table 2.2 continued on next page

[2]Tool dependent. Manual version control in Osate2.
[3]Tool dependent. No explicit library concept in Papyrus.

Feature	AADL	AutoFocus3	ArchJava & Java/A	Ptolemy II	UML & SysML		
Math. Foundation	informal, event-based	time-sync. FOCUS	informal, method calls	director based	informal, method calls	informal, event-based	parametric equations
Comp. Timing Classification	no	weakly, strongly causal time-sync.	no	directors: DE = timed, SDF = time-sync., PN = untimed	no	no	no
Simulation Time	yes, global clock	yes, global clock	no	yes, global, hierarchical clock	no	yes, global clock	no, yes in [KND+09]
Distribution	no	no	manual	no	manual	yes, extra effort for time sync.	solver dependent
Comp. Testing	trace analysis	table based	Java JUnit	interactive exploration	test spec + ODs + SDs	opp_test tool	SysML SDs and requirement diagrams
Determinism	yes	yes	yes	yes	yes	yes	yes
Black-box Behavior	partial	yes	yes	no	yes	yes	yes
White-box Behavior	partial	no	no yes	no	yes	yes	yes
Mocks	no	partial, modes	no yes	no	yes	no	no
Timed Behavior	partial	yes, sim. step = time	no	yes	no	yes	partial

Table 2.2 continued on next page

Feature	**AADL**	**AutoFocus3**	**ArchJava & Java/A**	**Ptolemy II**	**UML & SysML**		
Extensibility	mentioned, not documented	no	no	directors and actors	custom generators, handwritten code	plug-in mechanism	solver
Scheduler							
Default	internal	internal	no	directors	no	yes	yes
Custom Schedulers	no	no	no	directors	no	yes	no
Mult. Act. Sched.	no	no	no	yes	no	no	solver & SysML scheduler

Table 2.2: Overview of simulation features provided by the examined related work.

Chapter 3

MontiArc ADL

As discussed in Chapter 1, architecture description languages (ADLs) are rarely used in industrial practice due to several reasons. In Chapter 2, requirements for an ADL are derived which encounter the identified weaknesses and an overview of currently used ADLs and architectural programming languages (APLs) is given. It is shown that none of the regarded languages fulfills the derived requirements to the full extend and an extension of these languages is not promising.

On this account, the MontiArc language and corresponding tools have been developed from the scratch using the language workbench MontiCore [KRV06, Kra10]. Since MontiCore supports various kinds of language extension and composition mechanisms, it is well suited to realize an extendable ADL. Further, good concepts from the aforementioned ADLs have been integrated into MontiArc. Among them are the light-weight textual notation of the Architecture Analysis and Design Language (AADL) and Acme, architectural extensibility of xADL, the mathematical foundations of FOCUS, concepts of architectural programming from ArchJava and Java/A, and the flexible combination of different time domains from Ptolemy II.

The ADL MontiArc aims at modeling logical architectures of interactive, distributed systems. Its architectural style is based on the FOCUS framework [BS01]. Thus, MontiArc components correspond to distributed actors which asynchronously communicate with message passing over directed channels. Considering the requirements for architectural modeling which are given in the previous chapter, MontiArc is constructed with a small set of modeling elements which are based on the formal semantics taken from FOCUS. As a result, MontiArc is easy to learn, yet powerful enough to simulate the communication of architectural models. MontiArc's most fundamental features are:

- A simple core architectural style based on FOCUS.
- A modular and extendable language design to enable refinements and adaptions to tailor the architectural style to further domains.
- Suitability for distributed, collaborative, and agile modeling with a set of model analyses which can be executed without user interaction.
- A FOCUS based simulation of MontiArc models and automatic component tests (see Chapter 4).
- High degree of model-reusability with configurable and generic components as well as component libraries.
- Integrated modeling comfort functions.
- Compatibility to AADL [FGH06].

This chapter is structured as follows. First, the MontiArc ADL is introduced with an initial

example. Then, the concepts of architectural model elements provided by MontiArc and the simulation specific extension are discussed in Section 3.2 and Section 3.3. The concrete MontiArc language and its defining grammars are presented in Section 3.4. Context conditions that are used to analyse and validate MontiArc models are defined and explained in Section 3.5. Finally, a mapping to the popular AADL is proposed in Section 3.6.

3.1 A MontiArc Example

The syntax of MontiArc is introduced by means of an architectural model which defines the logical architecture of an automotive function. The component `LightCtrl` controls the interior light of a car and provides the following behavior that depends on received messages as well as time events:

- If an alarm is active, the light blinks with a configurable interval.
- It turns on the light if the status of the switch is *on*.
- It turns off the light if the status of the switch is *off*.
- If the switch status is *door dependent*, the state of the doors is determining its behavior. Consequently, if the door is opened, the light is turned on. If the door is closed, the light is turned off after a configurable delay.

A MontiArc component that defines the architecture of the light control function is depicted in Listing 3.1. The illustrated model contains the most frequently used elements of MontiArc. Similar to Java, components are organized in packages that represent the folder structure MontiArc models are located in (cf. l. 1). As shown in l. 2, MontiArc allows to import all types of a package at once using a star import. This way, all data as well as component types defined in package `ila.signals` can be referenced in an unqualified way within the current component definition. Direct type imports are also supported.

The depicted component definition contains three subcomponents that instantiate component types. An alarm check subcomponent (l. 22) interprets the alarm status of the car and causes the interior light to blink if an alarm is present. The door status is evaluated by another subcomponent that realizes a door dependent switching of the interior light (l. 23). These contradictory resulting requests are deliberated by an arbiter subcomponent to compute the resulting command for the interior light. The configurable component `DoorEval` is additionally configured with the value of parameter `fadeOutTime` (l. 23) which is a configuration parameter of component `LightCtrl` (l. 4). This way, the light fade-out time after closing a door is set to `fadeOutTime` time intervals and can be configured when instantiating the modeled system. Subcomponent `ac` is configured with the same parameter and an added value of 2 (l. 22). The type of the `Arbiter` subcomponent is, in contrast to the aforementioned subcomponents, defined as an inner component definition (ll. 14 – 20). Since the `autoinstantiate` feature is activated (l. 5), inner component `Arbiter` is automatically instantiated as subcomponent `arbiter`.

The component `LightCtrl` can only be accessed via messages on its incoming ports. If the name of a port is omitted, it is implicitly named after its type. Hence, its incoming ports are named `switchStatus`, `alarmStatus` and `doorStatus` (ll. 9 – 11). The computed result

```
1 package ila;                                                    MA
2 import ila.signals.*;
3
4 component LightCtrl[int fadeOutTime]  {
5   autoinstantiate on;
6   autoconnect port;
7
8   port
9     in SwitchStatus,
10    in AlarmStatus,
11    in DoorStatus,
12    out OnOffCmd cmd;
13
14  component Arbiter {
15    port
16      in SwitchStatus,
17      in BlinkRequest,
18      in OnOffRequest,
19      out OnOffCmd;
20  }
21
22  component AlarmCheck(fadeOutTime + 2) ac;
23  component DoorEval(fadeOutTime);
24
25  connect arbiter.onOffCmd -> cmd;
26 }
```

Listing 3.1: Component `LightCtrl` with its inner component definition `Arbiter` in textual MontiArc syntax. The contained subcomponents are automatically connected using the autoconnect feature.

will then be sent via an outgoing port of type `OnOffCmd` that is named `cmd` (l. 12). The ports of the contained subcomponents are connected to each other and to the outer ports via connectors.

Connectors are directed and always connect one sending port with an arbitrary number of receiving ports of compatible data types. Two ports can be connected if either their data types are identical or the data type of the receiver is a supertype of the sender's data type. The connector in l. 25 connects the outgoing port `onOffCmd` that belongs to subcomponent `arbiter` with the outgoing port `cmd`. Since the autoconnect feature of the MontiArc language is activated (l. 6), all other ports of the example are automatically connected. In MontiArc, communication is unidirectional. If a response is needed from a connected subcomponent, this has to be modeled as a feedback via another communication channel. MontiArc assumes asynchronous communication, which is also implemented in the MontiArc simulation (see Chapter 4), since it is better suited for modeling parallel and distributed computations.

3.2 Basic Architectural Model Elements

Considering *RQ1* and requirement *LRQ1*, MontiArc is designed to be simple on the one hand, so it can be used as an APL to teach and explore interactive systems. On the other hand, it is a complete and functional ADL that is capable to model and simulate distributed interactive systems in the style of an APL. To achieve this goal, MontiArc is split into two language components: an architectural base language and a simulation specific extension. The former named Architecture Diagram (ArcD) language comprises all basic architectural elements. That way, it provides a very compact notation for a subset of the most relevant modeling elements of the AADL with the same semantics and domain (see Section 3.6). The MontiArc language extends the ArcD language to add simulation and timing specific model information (see Section 3.4.2) and further adds comfort modeling features.

The basic elements of the ArcD language are, according to requirement *LRQ1.1*, component type definitions with ports that define their interfaces (requirement *LRQ1.2*) and subcomponent declarations that instantiate further component definitions. Subcomponents are part of the architectural configuration of a component type. The configuration is supplemented by connectors that unidirectionally connect ports of components and subcomponents (see requirement *LRQ1.3*). These model elements are presented in the following. Since the MontiArc language is an extension of the ArcD language, and consequently contains the same model elements, the term MontiArc is used synonymously for both languages.

3.2.1 Component Type Definition

Component type definitions (or short *components*) introduce new component types that are identified by unique qualified names (see requirement *LRQ1.1.2*) that provide an interface via which the component interacts with its environment (see Section 3.2.2).

In MontiArc, components can either be implemented directly (as *atomic components*) or defined to be a composition of other components. These *decomposed components* are hierarchically structured into further subcomponents (as for example `LightCtrl` in Listing 3.1) and thus have their behavior derived from the composition of their subcomponents. For atomic components, a reaction to incoming messages can be specified directly. Since the interfaces of atomic and decomposed components are their only interaction points with the environment, both component kinds can be equally treated in an architectural model. Thus, all subcomponents can be regarded as black boxes whose observable behavior conforms to their interface.

A simple definition of a component type in MontiArc syntax is given in Listing 3.2. After the declaration of the package, a component definitions starts with the keyword `component` followed by the component's name. The qualified name of the depicted component type is composed of the package (l. 1) and the component's name (l.3) to `ma.snippets.comp.A`. The component body embraces the model elements with curly brackets which define the component's interface and architectural configuration (see below).

MontiArc provides a *structural inheritance* mechanism that allows to define a component as an extension of another component type (see requirement *LRQ1.1.1*). The new type inherits the interface as well as the architectural configuration from the supercomponent. Thus, all ports, inner component type definitions, subcomponents, and connectors are inherited. An example is

```
1 package ma.snippets.comp;                                    MA
2 component A {                                                ...
3 }
```

Listing 3.2: Definition of component type `A`.

depicted in Listing 3.3. Similar to Java, component inheritance is introduced by the keyword `extends` followed by the type of the supercomponent. Thus, component `Ext` extends component `A`. Since the type of `A` has been imported (l. 1), the unqualified type name can be used in the extends clause (l. 2).

```
1 import ma.snippets.comp.A;                                   MA
2 component Ext extends A {                                    ...
3 }
```

Listing 3.3: Definition of component type `Ext` as an extension of component `A`.

MontiArc supports the definition of *configurable component types*. Such components define configuration parameters which represent variables with a certain data type. These parameters are used within the implementation of a component and values have to be assigned when a configurable component is instantiated. Consequently, parameters allow to adjust the behavior of a component to the current context (see requirement *LRQ1.1.1*). Configuration parameters in MontiArc are similar to a parametrized constructor of an object oriented class. An example for the definition of configurable components is depicted in Listing 3.4. Component `B` defines two configuration parameters `p1` and `p2` with the data types `int` and `String` (l.1). Configuration parameters are embraced by `[` and `]`. These parameters can be used within the architectural configuration of decomposed components and are directly passed to the implementation of atomic components (see Section 5.5).

```
1 component B[int p1, String p2] {                             MA
2 }                                                            ...
```

Listing 3.4: Definition of configurable component type `B`.

Generic component type definitions, or short *generic components*, can be parametrized with type parameters. These can be used as data types for the ports of a generic component or as type parameters in subcomponent declarations. This way, the concrete interface of a component can be adjusted to the current needs when instantiating a generic component as subcomponent (see requirement *LRQ1.1.1*). As depicted in Listing 3.5, a notation similar to Java generics is used. The type parameters `K` and `V` are directly attached to the component's name and are embraced by < and >. MontiArc also offers the possibility to restrict the upper bound of type parameters using the keyword `extends`. If a generic component with restricted type parameters is instantiated, the assigned types have to meet the defined upper bounds. Thus, it has to implement or be a subclass of the upper bounds type. Please note, if a generic component is also configurable, the list of configuration parameters is given after the list of type parameters.

```
1 component C<K extends Number, V> {                                    MA
2 }                                                                      ...
```

Listing 3.5: Definition of generic component type `C`.

3.2.2 Component Interface

Medvidovic and Tailor define a component interface as a set of interaction points between the component and its external environment [MT00]. MontiArc component interfaces are defined by a set of incoming and outgoing ports (see requirement *LRQ1.2*). Considering requirement *LRQ1*, a MontiArc port is defined as an interaction point that asynchronously and unidirectionally sends (outgoing) or receives (incoming) events respectively data messages of the port's *data type*.

The referenced data types of MontiArc ports can be defined using UML/P class diagrams [Rum11, Chapter 2], [Sch12, Section 3.1] or plain Java classes (see requirement *LRQ4.2*). Nevertheless, further type definition languages can be added to MontiArc if necessary (see requirement *LRQ4.1*). The needed extension method is described in Chapter 7. Beside the type, a port can also have a name. Determined by requirement *LRQ1.4*, naming of a port is optional as long as its type is solely used for this port in the scope of the component definition. In this case, an unnamed port is referenced by its unqualified type name (starting with a small letter). MontiArc does not define its own data type system but reuses the aforementioned existing type systems. In this way, types, which have already been defined in a class diagram or in the target language, need not be redefined in a specific type language but can be directly reused.

The interface definition of component `A` is depicted in Listing 3.6. It starts with the keyword `port` (l. 3) followed by a list of incoming and outgoing ports. The first port in l. 4 receives messages with data type `String` and has the name `string` (see above). The second port named `command` has the enumeration data type `Cmd`, which is imported in l. 1 and defined in the UML/P CD that is depicted in Listing 3.7. Please note that the syntax of UML/P CDs is

```
1 import ma.snippets.signals.Cmd;                                        MA
2 component A {                                                          ...
3   port
4     in String,
5     in Cmd command,
6     out Integer;
7 }
```

Listing 3.6: Interface definition of component type `A`.

```
1 package ma.snippets;                                                   CD
2 classdiagram signals {
3   enum Cmd { PULL, PUSH; }
4 }
```

Listing 3.7: Definition of data types using an UML/P CD.

completely defined in [Sch12, Section 3.1]. The third port named `integer` has the data type `Integer`. It can be used by the component to send `Integer` messages to its environment.

As already mentioned, generic type parameters can be used as port data types. An example is given in Listing 3.8. Type parameter `K` is used as data type from the incoming port `msgIn` (l. 3), type parameter `V` is used as data type from the outgoing port `msgOut` (l. 4). Please note that type parameters of generic components and configuration parameters of configurable components (see Section 3.2.1) are also part of a component's interface.

```
1 component C<K extends Number, V> {                    MA
2   port                                                 ...
3     in K msgIn,
4     out V msgOut;
5 }
```

Listing 3.8: Interface definition of generic component type `C`.

3.2.3 Architectural Configuration

Medvidovic and Tailor [MT00] define the configuration of an architecture as a connected graph of components and connectors that together describe the architectural structure of a system. Thus, the configuration of a component describes its decomposition into subcomponents, the connectors between the subcomponents, and the connectors that connect the interface of a component with its subcomponents.

Subcomponent Declarations

A MontiArc *subcomponent declaration* (or *subcomponent* in short) instantiates a component implementation and associates a local name. This allows to reuse components in the scope of another component (see requirement *LRQ1.1.1*). Similar to ports, naming of subcomponents is optional if the referenced component type is unique withing the current scope (see requirement *LRQ1.4*). Missing names are then derived from the referenced component type. If the instantiated component is a generic or configurable component, type parameters or configuration parameters are to be assigned during instantiation (see requirement *LRQ1.2*).

Listing 3.9 depicts three subcomponent declarations that start with the keyword `component`. Subcomponent `a` instantiates component type `A` (l. 2). In l. 3, two subcomponents `myB1` and `myB2` instantiate the configurable component type `B` (see Listing 3.4). The value `5` is assigned to the first parameter (`p1`) and the String `"foo"` is assigned to the second parameter (`p2`). Subcomponent `c` (l. 4) instantiates the generic component type `C` (see Listing 3.8) and assigns the types `Integer` and `String` to the type parameters `K` and `V`. In this way, the port `msgIn` of subcomponent `c` has the data type `Integer` and port `msgOut` the type `String`. Please note, `Integer` is a valid type for the restricted type parameter `K` since it is a subtype of `Number`.

```
1 component D {                                                    MA
2   component A;                                                   ...
3   component B(5, "foo") myB1, myB2;
4   component C<Integer, String>;
5 }
```

Listing 3.9: Subcomponent declarations in MontiArc syntax.

Connectors

Medvidovic and Tailor [MT00] define a *connector* as an architectural element that is used to model interactions between components together with rules that govern those interactions. In MontiArc, a connector models a communication channel that unidirectionally and immediately transmits data messages from a sender to a receiver without message loss (see requirement *LRQ1.3*). Rules that define the valid definition of connectors are given in Section 3.5.2.

Exemplary MontiArc connectors are depicted in Listing 3.10. A connector starts with the keyword `connect` and consists of a source and a list of targets. Sources and targets can be either a port from the current component definition or a port from a subcomponent. For example the source of the connector in l. 5 is the incoming port `sIn` (l. 2) and the target is the incoming port `sIn` of subcomponent `myB1`. It is also possible to define a list of targets for a single connector (l. 7). In this way, messages emitted by the sender are transmitted to all connected receivers. It is further possible to attach connectors directly to outgoing ports of instantiated subcomponents (see l. 8). Such a connector is embraced by square brackets and its source references an outgoing port of the corresponding subcomponent. Thus, the depicted connector connects port `integer` of subcomponent `myExt` with the outgoing port `iOut` (l. 4).

```
1 port                                                             MA
2   in String sIn,                                                 ...
3   out String sOut,
4   out Integer iOut;
5 connect sIn -> myB1.sIn;
6 connect c.msgOut -> sOut;
7 connect myB1.sOut -> a.string, myExt.string;
8 component Ext myExt [integer -> iOut];
```

Listing 3.10: Connector definitions from component `D` in MontiArc syntax.

Inner Component Type Definitions

Inner component type definitions, or short *inner components*, are defined within an outer component. Similar to a local private class in Java, inner components can only be referenced within the scope of the component definition in which they are defined (see requirement *LRQ1.1.4*). In this way, static architectures with singleton-like components can be modeled easily. Inner components are defined with the same syntax just like regular component definitions. Examples

for the definition of inner components are depicted in Listing 3.11. Component `F` contains two inner component definitions `InnerA` (ll. 2f) and `InnerB` (ll. 6f). Both can only be used within the scope of component `F`. The former is instantiated as two subcomponents `ia1` and `ia2` in l. 4. The latter is automatically instantiated as subcomponent `ib1` along with the inner component definition. This is achieved by defining the name of the subcomponent directly after the inner component's type name (l. 6). Of course, inner component definitions can themself contain further inner component definitions.

```
1 component F {                                                    MA
2   component InnerA {                                             ...
3   }
4   component InnerA ia1, ia2;
5
6   component InnerB ib1 {
7   }
8 }
```

Listing 3.11: Inner component type definitions in MontiArc syntax.

3.3 Advanced Architectural Model Elements

The language MontiArc is defined as an extension of the ArcD language. It preserves all aforementioned language concepts and adds:

- declaration of the component's timing,
- implicit model completion for connections (autoconnect),
- implicit subcomponent instantiation of inner component definitions, and
- constraints on the component's behavior or state.

3.3.1 Component Timing

The *timing* of components can be defined as part of the component type definition in MontiArc (see requirement *LRQ1.1.3*). The available time domains are `instant`, `delayed`, `untimed`, `causalsync`, and `sync`. The definition of these domains is given in Section 4.4 on page 105. If the timing is not explicitly declared within a component type definition, the component is assumed to have an instant timing . An example for the explicit selection of a time domain is depicted in Listing 3.12. The keyword `timing` introduces the time domain selection clause followed by the name of the selected domain (l. 2). In this way, component J has a causal synchronous timing.

3.3.2 Autoconnect

MontiArc allows to automatically complete component definitions with connectors according to some predefined rules (see requirement *LRQ1.5*). The currently available *connector completion strategies* are `port` and `type`. Former connects ports with matching name and compatible

```
1 component J {                                                    MA
2   timing causalsync;                                             ...
3 }
```

Listing 3.12: Selection of the causal synchronous time domain for component J.

type. The latter connects ports with matching types. The parameter off disables automatic connection of ports. Component G, which is depicted in Listing 3.13, enabled the *type* completion strategy (l. 2). In this way, all shown ports are automatically connected since the data types are compatible. If the port completion strategy would have been used for the same component, none of these ports would have been connected since the names do not match.

```
1 component G {                                                    MA
2   autoconnect type;                                              ...
3   port
4     in String sOuter,
5     out Integer iOuter;
6   component InnerA ia {
7     port
8       in String sInner,
9       out Integer iInner;
10  }
11 }
```

Listing 3.13: Using the autoconnect statement.

3.3.3 Autoinstantiate

In many cases it is the modelers intention to instantiate an inner component definition directly as a subcomponent. To *automatically instantiate* subcomponents together with their definition, the modeler can enable the autoinstantiate concept on the parent's component level (see requirement *LRQ1.1.4*). If this feature is enabled, all inner component definitions that are neither generic nor configurable are automatically instantiated as a subcomponent. Please note that this mechanism only instantiates inner components as subcomponents if the resulting subcomponent name is not already in use. A warning is raised if inner component definitions cannot be instantiated automatically. This way, it is indicated that the inner component definition is yet not used. An example for the automatic instantiation of an inner component is depicted in Listing 3.14. The autoconnect feature is switched on (l. 1) which leads to to an automatic instantiation of inner component InnerC (ll. 4 – 7) as subcomponent innerC. In this way, the subcomponent can be immediately used as source of the connector depicted in l. 8.

3.3.4 Constraints

Constraints can be defined for components and written in almost any external constraint language. Using MontiCore's language embedding features [GKR+08], the MontiArc language is

```
1 autoinstantiate on;                                          MA
2 port                                                         ...
3   out String sOuter;
4 component InnerC {
5   port
6     out String sInner;
7 }
8 connect innerC.sInner -> sOuter;
```

Listing 3.14: Using the `autoinstantiate` statement.

currently configured to use Java and OCL expressions, which have to evaluate to Boolean expressions, to define constraints (see requirement *LRQ1.6*). Exemplary constraints are depicted in Listing 3.15. A Java constraint has to start with the language identifier `java` (l. 4). Since OCL is the default constraint language, the language identifier `ocl` is optional (l. 8). After the language identifier, the keyword `inv` is expected followed by the name of the constraint, a colon, and the concrete constraint. Java constraints are embraced by curly brackets (ll. 4 – 7), OCL constraints can be directly integrated (l. 9). All constraint definitions end with a semicolon (ll. 7, 9). The complete language definition of the used OCL dialect is given in [Rum11, Sch12].

```
1 port                                                         MA
2   in String sIn,                                             ...
3   out String sOut;
4 java inv numberOfMsgs: {
5   sIn.stream.getUntimedHistory().size() ==
6     sOut.stream.getUntimedHistory().size() - 1;
7 };
8 inv msgValues:
9   sOut == sIn@pre;
```

Listing 3.15: The definition of Java and OCL constraints.

3.4 MontiArc Language Definition

The ADL MontiArc is developed using the language workbench MontiCore. After an introduction to the concepts of MontiCore, the involved languages, their hierarchy, and their relations are described in Section 3.4.2. Finally, a detailed presentation of MontiArc's defining grammars is given. Please note that this and the following section are a revised version of the technical report *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems (AIB-2012-03)* [HRR12] that has been authored together with Jan O. Ringert and Bernhard Rumpe.

3.4.1 Foundations: MontiCore 3

MontiArc is developed using the DSL framework *MontiCore* [GKR+06, GKR+08, KRV08, Kra10, KRV10] according to the guidelines presented in [KKP+09]. MontiCore aims at supporting agile language development of domain specific languages (DSLs) [Cza04] and the according language processing tools such as generators or analysis workflows.

This brief introduction to MontiCore only contains a small part of the language and tool features MontiCore offers. Nevertheless, it describes the concepts needed to understand the following sections. A more detailed and complete description of the MontiCore language and base infrastructure is given in [Kra10], the compositional symbol table concepts are given in [Völ11, Chapter 7] and [HLMSN+15]. For a detailed description of the code generator framework it is referred to [Sch12, Section 7.4].

MontiCore takes a grammar artifact as input, the grammar format is based on the Extended Backus-Naur Form (EBNF) and Antlr [www13a], and generates parser, lexer, and Java classes to represent the abstract syntax tree (AST) of input models. This way, a grammar defines the concrete as well as the abstract syntax of a language.

To ease the development process of DSLs and increase its efficiency, MontiCore supports three different kinds of language reuse [LNPR+13, HLMSN+15]. *Language aggregation* allows to combine multiple languages into a collection of languages. The contained languages are allowed to reference model elements defined in any language contained in this collection. *Language inheritance* allows to extend an existing language definition. The sub-language inherits all productions from the parent-language. This mechanism allows language refinement in an object oriented manner by overwriting existing language productions or adding new productions to increase the expressiveness of a language. *Language embedding* allows embedding of single productions of a language into another language. This way, e.g., constraint languages like the OCL/P [Rum11, Chapter 3] can be reused in different contexts.

Beside the language generator, MontiCore serves a base infrastructure to support the uniform development of language processing tools. This so called *DSLTool-Framework*, which has been originally documented in [Kra10, Chapter 9], mainly contains:

- An infrastructure for uniform error handling.
- A generator development infrastructure [Sch12, Section 7.4] based on the Freemarker template engine [www13b].
- Tools and infrastructure for the development of compositional symbol tables [Völ11, Chapter 7].
- A context condition infrastructure to unify model checks [Völ11, Chapter 8], [Sch12, Section 7.3].
- Tools for visitor [GHJV95] and workflow based language and model processing.
- Basic languages for, e.g., literal or type definition [Sch12, Section 3.8].
- Language definitions and tools to reuse and process more powerful languages such as Java.

To understand the simplified grammars discussed in Section 3.4.3 and Section 3.4.4, some basic concepts of MontiCore grammars are introduced. These concepts are explained by means of a simple automaton language which is defined by the grammar depicted in Listing 3.16.

MG

```
1 grammar Automaton {
2   interface AutElement;
3
4   Automaton =
5     "automaton" Name "{" (Elements:AutElement)* "}" ;
6
7   State implements AutElement =
8     "state" Name
9     (("<<" Initial:["initial"] ">>") |
10    ("<<" Final:["final"] ">>"))*
11    ( ("{" (Elements:AutElement)* "}") | ";") ;
12
13  Transition implements AutElement =
14    From:Name "-" Activate:Name ">" To:Name ";" ;
15 }
```

Listing 3.16: A simplified MontiCore grammar which defines an automaton language (based on [GKR+06]).

A MontiCore grammar contains a set of productions that consist of a left-hand side (LHS) which is separated by a = from a right-hand side (RHS). Productions define the nonterminals of the language. Thus, the automaton language contains three productions which define three nonterminals `Automaton`, `State`, and `Transition`. For each nonterminal MontiCore derives an AST Java class that represents the nonterminal in the abstract syntax. A special case is an interface nonterminal that is defined by a production which consists of a LHS only (see interface `AutElement`). A LHS of a production defines the name of the nonterminal as its type. If the LHS of a production implements an interface, the defined nonterminal can be used in all RHSs that contain the interface nonterminal. Consequently, the nonterminals `State` and `Transition`, whose defining productions implement the aforementioned interface (cf. ll. 7, 13), can be used within the RHSs of the productions that define the nonterminals `Automaton` (cf. l. 5) and `State` (cf. l. 11).

A RHS defines the mapping rule of the nonterminal given by terminals and nonterminals. Alternatives are separated by a `|` and optional elements are annotated with a `?`. Thus, a state can have the markers `initial` or `final` (see ll. 9f). The Kleene-Star $\star$ marks elements to occur arbitrary often, a + defines that an element has to occur at least once. Consequently, a state can not be marked at all or it can be marked with both markers.

Terminals respectively keywords of a language are bordered by quotes (e.g., `"state"` in l. 8). A keyword is part of the concrete syntax only and is not represented within its AST. If a keyword should be present in the AST, it has to be additionally bordered by squared brackets. Then MontiCore derives a Boolean attribute with the name of the keywords in the AST class that represents the nonterminal. Thus, the markers of a state, i.e. `initial` and `final`, are available in the AST as well (cf. ll. 9f). Please note that `variableName:Nonterminal` is an extension to normal grammars, where `Nonterminal` is the nonterminal (type) and `variableName` is the name of the associated variable in the abstract syntax that usually also codes

the form of use. Thus, a nonterminal can be used more than once within the RHS of a production. To distinguish multiply used nonterminals in the AST, a local variableName has to be assigned to each occurrence. In the example, it allows to distinguish between the source and target state of a transition (cf. l. 14).

3.4.2 Language Structure

As explained in the previous section, MontiCore provides several language extension mechanisms. Language inheritance allows to reuse existing grammars as a base of a new language. The language hierarchy of MontiArc, which uses MontiCore's inheritance mechanism, is shown in Figure 3.17.

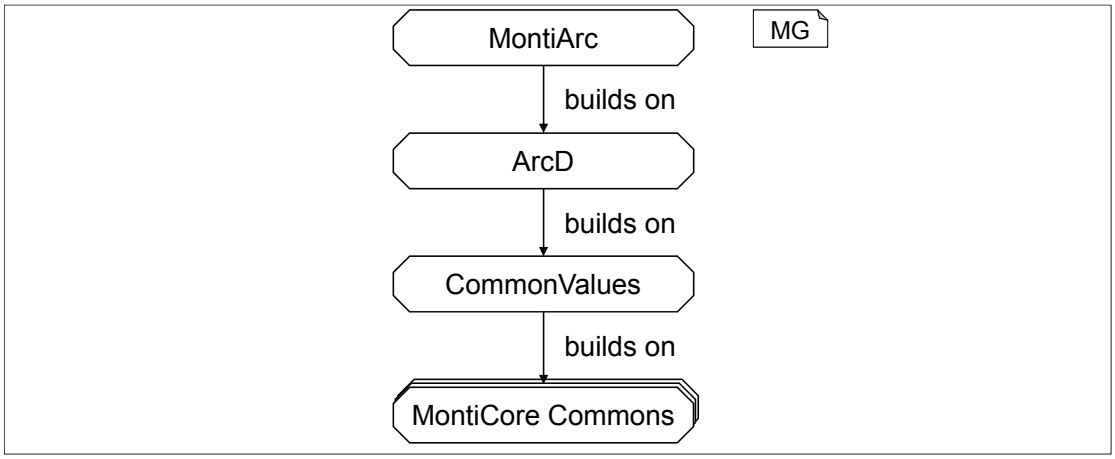


Figure 3.17: MontiCore grammar hierarchy of the MontiArc language.

- `MontiCore Commons` is a collection of grammars (Literals, Types, and Commons) that provides common modeling language artifacts such as stereotypes, cardinalities, modifiers, as well as productions that define nonterminals for type references and literals. A detailed description of the used base languages is given in [Sch12, Section 3.8].
- `CommonValues` is a grammar that provides productions which define nonterminals for value assignments. These nonterminals are aggregated under a single nonterminal interface. Reusing this interface allows to use several value kinds at a single position of a RHS. These kinds are literals (inherited from MontiCore Commons), variables respectively variable names, return values of method calls, constants, enumeration values, arrays, and simple computations.
- `ArchitectureDiagram` defines the Architecture Diagram (ArcD) language which provides the basic architectural elements presented in Section 3.2.
- `MontiArc` extends ArcD and adds language elements which are tailored to the event-based simulation of the modeled distributed systems (see Section 3.3).

Simplified MontiCore grammars of the ArcD and MontiArc languages are explained in the following two subsections. Complete grammar definitions are given in the appendix in Listing C.1 on page 297 and Listing C.2 on page 302.

3.4.3 Architecture Diagram Grammar Walk-Through

By using MontiCore's compilation unit concept (see [GKR+06, Section 5.1]), component type definitions are organized in packages and build the root elements of ArcD models. The production that defines nonterminal `ArcComponent`, which is shown in Listing 3.18, defines the structure of a component type definition. It can be annotated with a stereotype followed by the keyword `component` and a component type name. The qualified type name of a component definition corresponds to its package appended with its dot-separated name. The optional `instanceName` can be used to create a subcomponent declaration along with the definition of an inner component type. For top-level component definitions, the usage of an instance name is forbidden (cf. context condition B2 in Section 3.5.1). The RHS further contains `ArcComponentHead` and `ArcComponentBody` nonterminals.

```
1 ArcComponent implements ArcElement =                                MG
2   Stereotype?                                                       ...
3   "component" Name (instanceName:Name)?
4   ArcComponentHead ArcComponentBody;
```

Listing 3.18: `ArchitectureDiagram.mc`: Definition of nonterminal `ArcComponent` for component type definitions.

The production that defines nonterminal `ArcComponentHead` is shown in Listing 3.19. It provides the optional definition of `TypeParameters` (cf. l. 2) that are used to define generic type variables. These variables can serve as port data types in the scope of the component body. A list of `ArcParameters` is enclosed by squared brackets (cf. l. 3). These are used to define configuration parameters with a type and a name which are visible in the scope of the component definition (cf. ll. 6f). Finally, a component type can extend a supercomponent. For this, the supercomponent type name is given after the `extends` keyword (cf. l. 4).

```
1 ArcComponentHead =                                                  MG
2   TypeParameters?                                                   ...
3   ("[" ArcParameter* "]")?
4   ("extends" ReferenceType)?;
5
6 ArcParameter =
7   Type Name;
```

Listing 3.19: `ArchitectureDiagram.mc`: Definition of nonterminals `ArcComponentHead` and `ArcParameter`.

Listing 3.20 depicts the production that defines nonterminal `ArcComponentBody` which defines the syntax of of a component body. It contains arbitrary many `ArcElements` that

are parenthesized by curly brackets. `ArcElement` is an interface for productions that define nonterminals of architectural elements and can occur in a component type definition. Therefore, the inner structure of a component type is given by a set of `ArcElements`. As depicted in Listing 3.18 in line 1, the component nonterminal itself implements this interface. In this way, inner components can be defined. To extend this language with more elements that can be part of a component type, new productions that implement this interface can be defined in a subgrammar. A structured method for language extension is given in Chapter 7.

```
1 ArcComponentBody =                                                    MG
2   "{"                                                                 ...
3     ArcElement*
4   "}";
```

Listing 3.20: `ArchitectureDiagram.mc`: Definition of nonterminal `ArcComponentBody`.

The production for nonterminal `ArcInterface`, which defines the syntax of a component interface, is given in Listing 3.21. After an optional stereotype and the keyword `port` a list of ports is given. A port (cf. ll. 5 – 8) can have a stereotype. After the port's direction, `in` is used for incoming and `out` for outgoing ports, the port's data type and an optional name are given.

```
1 ArcInterface implements ArcElement =                                  MG
2   Stereotype?                                                         ...
3   "port" (ArcPort)* ";";
4
5 ArcPort =
6   Stereotype?
7   ("in" | "out")
8   Type Name?;
```

Listing 3.21: `ArchitectureDiagram.mc`: Definition of nonterminals `ArcInterface` and `ArcPort`.

The internal structure of decomposed components is given by subcomponents. The syntax of a subcomponent declaration is defined by the production `ArcSubComponent` that is shown in Listing 3.22. After an optional stereotype and the keyword `component`, the type of the subcomponent is given. This is a reference to another component type definition. An optional list of arguments is parenthesized by round brackets. These arguments are used to set configuration parameters of referenced configurable components. It can be any `Value` as defined in the *CommonValues* supergrammar[1]. Hence, any literals, variable names, constants, enumeration values, arrays, method calls, or simple computations can be used as a subcomponent argument.

[1] HTML grammar documentation of the CommonValues DSL can by found at `https://sselab.de/lab2/public/web/MontiCore/grammardoc/common-values/1.9.0/index.html`.

To create more than one subcomponent declaration or to assign an explicit name, an optional list of instances is used (cf. l. 6).

```
1 ArcSubComponent implements ArcElement =                                MG
2   Stereotype?                                                           ...
3   "component"
4   ReferenceType
5   ("(" arguments:Value* ")" )?
6   (ArcSubComponentInstance* )? ";";
```

Listing 3.22: `ArchitectureDiagram.mc`: Definition of nonterminal `ArcSubComponent`.

The production that defines the nonterminal `ArcSubComponentInstance` is depicted in Listing 3.23. It has a name and an optional colon separated list of simple connectors parenthesized by squared brackets. Simple connectors (cf. ll. 4–7) directly connect outgoing ports of the corresponding subcomponent declaration with one or more target ports. The optional stereotype is followed by the connectors source and a list of targets.

```
1 ArcSubComponentInstance =                                              MG
2   Name ("[" ArcSimpleConnector (";" ArcSimpleConnector)* "]")?; ...
3
4 ArcSimpleConnector =
5   Stereotype?
6   source:QualifiedName "->" targets:QualifiedName
7                        ("," targets:QualifiedName)*;
```

Listing 3.23: `ArchitectureDiagram.mc`: Definition of nonterminal `ArcSubComponentInstance` which allows to explicitly name subcomponents and optionally associate simple connectors.

The general form to model communication paths is given by connectors. The syntax of nonterminal `ArcConnector` is defined by the production given in Listing 3.24. After an optional stereotype and the keyword `connect`, the `source` of the connector is given by a qualified name. After an arrow `->` one or more comma separated `targets` are given. Source or target of a connector can be either a port of the current component, a name of a subcomponent declaration, or a port that belongs to a subcomponent declaration. In the last case, the port is qualified by the name of the subcomponent to which it belongs.

```
1 ArcConnector implements ArcElement=                                    MG
2   Stereotype?                                                           ...
3   "connect" source:QualifiedName "->"
4   targets:QualifiedName ("," targets:QualifiedName)* ";";
```

Listing 3.24: `ArchitectureDiagram.mc`: Definition of nonterminal `ArcConnector`.

The ArcD language does not provide productions to define behavior of components. Therefore, an external nonterminal `BehaviorEmbedding` is included which allows to embed arbitrary behavior implementation languages. The production of nonterminal `ArcComponentImplementation` implements `ArcElement` as depicted in Listing 3.25 (l. 1). After the keyword `implementation`, the language identifier `lng` defines which concrete language is used for the following implementation. The implementation is embraced by curly braces. This nonterminal allows to implement component behavior using, e.g., statecharts (SCs), Java, or any other action language directly within the component. Details about language embedding are given in [Kra10, Section 4.2] and [HLMSN+15]. A method to extend the syntax of the ArcD or MontiArc language is discussed in Section 7.3.

```
1 ArcComponentImplementation implements ArcElement =                    MG
2   "implementation" lng:Name "{"                                       ...
3     BehaviorEmbedding
4   "}";
```

Listing 3.25: `ArchitectureDiagram.mc`: External behavior implementation embedding in ArcD.

3.4.4 MontiArc Grammar Walk-Through

MontiArc extends the ArchitectureDiagram language in two ways. First, to constrain the behavior and the state of a component, MontiArc adds language elements for constraints defined in OCL/P [Rum11, Chapter 3] or Java. This is shown in Listing 3.26. After the keyword `inv` the name of the constraint is given followed by the external nonterminal `InvariantContent`.

```
1 MontiArcInvariant implements ArcElement =                             MG
2   "inv" Name ":" InvariantContent ";";                                ...
```

Listing 3.26: `MontiArc.mc`: Definition of nonterminal `MontiArcInvariant`.

Second, it extends the language with configuration elements that have to implement the interface `MontiArcConfig`. This interface extends interface `ArcElement` that is inherited from the ArcD language. This way, a nonterminal that implements interface `MontiArcConfig` can be used within the body of a component definition. Further, configuration and regular elements can be distinguished in the AST.

The `autoconnect` statement represented by nonterminal `MontiArcAutoConnect` (cf. Listing 3.27) controls the autoconnect behavior of the component. The following modes are available:

- `type` automatically connects all ports with the same type name,
- `port` automatically connects all ports with the same port name if the port types are compatible, and
- `off` (default) turns auto connect off.

```
1 MontiArcAutoConnect implements MontiArcConfig =                    MG
2   "autoconnect" Stereotype?                                        ...
3   ("type" | "port" | "off") ";";
```

Listing 3.27: `MontiArc.mc`: Definition of nonterminal `MontiArcAutoConnect`.

Auto-instantiation is used to automatically create instances of inner component types if these are not explicitly declared as subcomponents. Listing 3.28 contains the production defining the syntax of this feature. After the keyword `autoinstantiate` and an optional stereotype, the mode is chosen using `on` or `off`, where `off` is the default case.

```
1 MontiArcAutoInstantiate implements MontiArcConfig =                MG
2   "autoinstantiate" Stereotype?                                    ...
3   ("on" | "off") ";";
```

Listing 3.28: `MontiArc.mc`: Definition of nonterminal `MontiArcAutoInstantiate`.

The definition of the nonterminal `MontiArcTiming`, which is used to denote which timing domain a component belongs to, is shown in Listing 3.29. After the keyword `timing` and an optional stereotype, the timing domain of the component can be selected. The available classifications are described in Section 4.4.

```
1 MontiArcTiming implements MontiArcConfig  =                        MG
2   "timing" Stereotype?                                             ...
3   (["instant"] | ["delayed"] | ["untimed"] | ["causalsync"] |
4       ["sync"]) ";";
```

Listing 3.29: `MontiArc.mc`: Definition of nonterminal `MontiArcTiming`.

3.5 Context Conditions

In MontiArc, quite a number of context condition checks are implemented in order to verify whether a MontiArc model is well-formed. Further, context conditions support the modeler with feedback about detected problems (see requirement *LRQ2.3* and *LRQ2.2*). These context conditions are grouped into general-, connections-, referential integrity-, conventions-, and code generation conditions. The following sections list these conditions and explain them by means of descriptive examples.

3.5.1 General Conditions

To define a concept for the visibility of identifiers, namespaces are introduced to MontiArc that define areas in a model in which names are managed together (cf. [Kra10, Völ11]). These identifiers are names of ports, subcomponent declarations, generic type variables, configuration

parameters, and constraint definitions. In MontiArc, it is distinguished between two different kinds of namespaces. A *component namespace* contains identifiers that are declared within a component type definition. Such a namespace is not hierarchical. Hence, identifiers defined in a top level namespace are not imported into a contained component namespace. In contrast, a *constraint namespace*, which is contained in a component namespace, imports all names that are defined within its parent namespace. A constraint namespace can also contain a hierarchical namespace structure according to the language that is used to define the constraint.

An example for namespaces, identifiers, and their visibility is given in Figure 3.30. The shown component type definition `FilterDelay` contains three namespaces. While identifier definitions are underlined with a solid line, references to these definitions are marked with a dashed line. The top-level namespace belongs to the component type definition itself. It contains the identifiers of the configuration parameter `fil`, the port names `inData` and `delayedAndFiltered`, the subcomponent declaration `f` and `del`, as well as the constraint `isFiltered` that is defined using an OCL/P expression. Please note that using the optional instance name while defining an inner component type will automatically declare a subcomponent with the used instance name (cf. Section 3.4.3). Since this subcomponent is declared in the parent component of the inner component definition, its identifier also belongs to the parents namespace.

The parent namespace `FilterDelay` contains another component namespace that belongs to the inner component type definition `Delay`. All identifiers within this namespace are colored gray. The port name `inData` is still unique since identifiers of the parent namespace, that also contains this name, are not imported.

Namespaces of constraints import identifiers of their parent namespace. On that account the

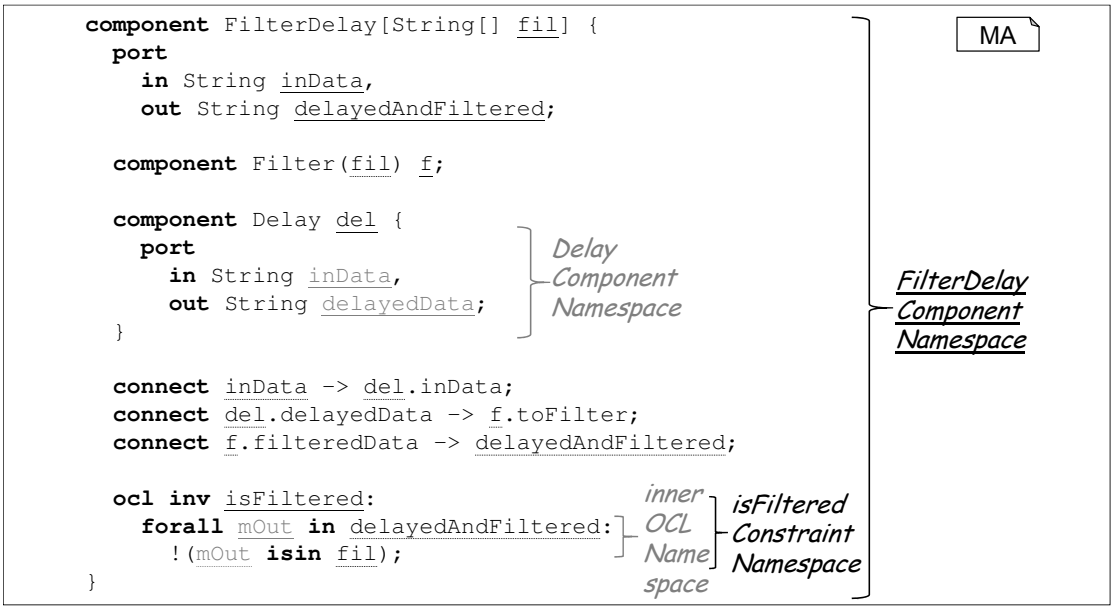


Figure 3.30: Namespaces and identifier declarations in MontiArc with embedded OCL/P constraints.

port name `delayedAndFiltered` as well as the parameter name `fil` can be used inside the namespace of constraint `isFiltered`. Since a `forall` construct opens a new namespace in the OCL/P language, it also has a hierarchical structure denoted by the `inner OCL` namespace. Details about the concrete namespace implementation are given in Section 5.2.3.

B1: All names of model elements within a component namespace have to be unique.

To clearly identify each model element, all names within a component namespace have to be unique. This holds for port names, subcomponent declaration names, generic type parameter names, configuration parameter names, and names of constraints. Listing 3.31 contains two violations of this condition. First, configuration parameter `fil` has the same name as one incoming port (see ll. 1, 3). Second, the subcomponent declaration `del` and a constraint have the same name (cf. ll. 8, 13).

```
1 component FilterDelay[String[] fil] {                                MA
2   port
3     in char[][] fil, ⊗// 'fil' already declared in l. 1
4     in String inData,
5     out String delayedAndFiltered;
6   component Filter(fil) f;
7
8   component Delay del {
9     port
10      in String inData,
11      out String delayedData;
12  }
13  ocl inv del: ⊗// 'del' already declared in l. 8
14    forall mOut in delayedAndFiltered:
15      !(mOut isin fil);
16 }
```

Listing 3.31: B1: Violation of context condition B1 by using names more than once in a namespace.

B2: Top-level component type definitions do not have instance names.

The optional instance name of component type definitions (cf. Section 3.4.3) is used to create a subcomponent declaration along with the definition of an inner component type. The created subcomponent then belongs to the parent component type. Root component types do not have a parent. Therefore, using an instance name for a root component type definition will result in a not assignable subcomponent. Hence, the usage of instance names for root component definitions is forbidden.

In Listing 3.32, the component definition `ABPSenderComponent` has the instance name `mySenderComp` which is not allowed. For the inner component definition `Sender` this concept is used to create a subcomponent named `innerSender` along with the definition of the inner component type.

```
1 component ABPSenderComponent mySenderComp { // instance          MA
2   component Sender innerSender {             // name for          ...
3   }                                          // root def.
4 }                                            // forbidden
```

Listing 3.32: B2: Instance names of component definitions.

3.5.2 Connections

The context conditions within this group validate whether connections within MontiArc models are well formed.

CO1: Connectors may not pierce through component interfaces.

Qualified sources and targets of a connector consist of two parts. The first part is a name of a subcomponent, the second part is a port name. Listing 3.33 contains the definition of component types `A_B_Filter` (cf. ll. 1–11) and `Filter` (cf. ll. 12–17). The former contains two subcomponent declarations of the latter.

The connector shown in l. 8 connects port `msgIn` of `A_B_Filter` with port `msgs` of subcomponent `af`. Since this port is part of the interface of `af`'s type, this connector is valid. The same holds for the second connector in l. 9 which connects the output of `bf` to the outgoing port `msgOut`. The target of the third connector shown in l. 10 is port `inData` of subcomponent `d`. This subcomponent belongs to component type `Filter` which is instantiated as subcomponent `bf`. Since subcomponent declarations are encapsulated and can only be accessed indirectly via their connected ports, subcomponent `d` is not visible in the scope of `A_B_Filter` and must not

```
1  component A_B_Filter {                                          MA
2    port                                                          ...
3      in String msgIn,
4      out String msgOut;
5    component Filter('a') af;
6    component Filter('b') bf;
7
8    connect msgIn -> af.msgs;
9    connect bf.filteredMsgs -> msgOut;
10   connect af.filteredMsgs -> bf.d.inData; // d not visible
11 }
12 component Filter[char f] {
13   port
14     in String msgs,
15     out String filteredMsgs;
16   component Delay(1) d;
17 }
```

Listing 3.33: CO1: Qualified sources and targets of connectors.

be used as a target of a connector.

CO2: A simple connector's source is an outgoing port of the referenced component type and is therefore not qualified.

A source of a simple connector always has to be an outgoing port of the subcomponent's component type. Therefore, a qualification is not needed since the port is implicitly qualified using the name of the associated subcomponent.

The first simple connector in line 6 of Listing 3.34 connects outgoing port `filteredMsgs` of subcomponent `af` with the incoming port `msgs` of subcomponent `bf` and is valid. The source of the second connector in l. 8 contains the subcomponent's name `bf` as an additional qualifier and is therefore invalid. Please note that the contained subcomponents instantiate component type `Filter` which is defined in Listing 3.33.

```
1 component A_B_Filter {                                          MA
2   port                                                          ...
3     in String msgIn,
4     out String msgOut;
5   component Filter('a') af
6     [filteredMsgs -> bf.msgs];
7   component Filter('b') bf
8     [bf.filteredMsgs -> bf.msgs]; // source is qualified
9 }
```

Listing 3.34: CO2: Correct and invalid sources of simple connectors.

CO3: Unqualified sources or targets in connectors either refer to a port or a subcomponent in the same namespace .

If sources or targets of a connector are unqualified, then they must refer to a port or a subcomponent name declared in the scope of the current component type definition. If a name of a subcomponent is used, all yet unconnected ports of the subcomponent with a compatible type are connected.

For example the first connector given in Listing 3.35 in l. 8 automatically resolves incoming port `msgs` of subcomponent `af` as the target of the connector since its type fits to the type of the connector's source. The second connector given in l. 9 connects all compatible outgoing ports of subcomponent `af` with all compatible incoming ports of subcomponent `bf`. The third connector is invalid since its target `cf` is neither the name of a port nor a subcomponent. Instead, `cf` corresponds to a configuration parameter. Finally, the fourth connector in l. 11 connects one compatible outgoing port of subcomponent `bf` with the outgoing port `msgOut`. This, however, is only possible if a unique compatible port can be resolved. If more than one compatible port is found, no connections are created and a warning is emitted.

```
1 component A_B_Filter[int cf] {                                    MA
2   port
3     in String msgIn,
4     out String msgOut;
5   component Filter('a') af;
6   component Filter('b') bf;
7
8   connect msgIn -> af;
9   connect af -> bf;
10  connect af -> cf; // cf is neither a port nor a subcomponent!
11  connect bf -> msgOut;
12 }
```

Listing 3.35: CO3: Using unqualified sources and targets in connectors.

3.5.3 Referential Integrity

Context conditions within this group validate the integrity of references within a MontiArc model. This is especially needed since the MontiArc AST only contains the name of referenced elements and does not contain a link to the corresponding referenced AST object.

R1: Each outgoing port of a component type definition is used at most once as target of a connector.

In MontiArc, the sender of a message or signal is always unique for the receiver (see requirement *LRQ1.3*). Hence, every receiving port only receives signals from a unique sender, while a sender can transmit its data to more than one receiver. Therefore, outgoing ports of a component type definition are used at most once as a target of a connector.

In Listing 3.36, the component type definition `A_B_Filter` violates this condition. The outgoing port `msgOut` is used as a target of the simple connector given in l. 7 and also as a target of the connector given in l. 11. A unique sender cannot be identified since the sender

```
1 component A_B_Filter {                                            MA
2   port
3     in String msgIn,
4     out String msgOut;
5
6   component Filter('a') af
7     [filteredMsgs -> msgOut]; // ambiguous sender
8   component Filter('b') bf;
9
10  connect msgIn -> af.msgs, bf.msgs;
11  connect bf.filteredMsgs -> msgOut; // ambiguous sender
12 }
```

Listing 3.36: R1: Ambiguous senders of connectors that have the same port as target.

could be the outgoing port of subcomponent `af` or the outgoing port of subcomponent `bf`. In contrast, a sender of a connector can transmit its messages to more than one receivers. Hence, the connector given in l. 10 is valid.

R2: Each incoming port of a subcomponent is used at most once as target of a connector.

As already discussed in the previous context condition, the sender of a message is always unique for a receiver. Incoming ports of subcomponents can be used as receivers in a connector and must therefore be used at most once as a receiver in the context of a component type definition. In Listing 3.37 this context condition is violated by the connectors given in ll. 9–10. The incoming port `msgs` of subcomponent `af` is used twice as a target.

```
1 component A_B_Filter {                                                    MA
2   port
3     in String msgIn,
4     out String msgOut;
5
6   component Filter('a') af;
7   component Filter('b') bf;
8
9   connect msgIn -> bf.msgs,  af.msgs; ⊗ // ambiguous sender
10  connect bf.filteredMsgs -> af.msgs; ⊗ // ambiguous sender
11  connect af.filteredMsgs -> msgOut;
12 }
```

Listing 3.37: R2: Ambiguous senders of connectors that have the same port of a subcomponent as target.

R3: Full qualified subcomponent types exist in the named package.

If a qualified component type is used as the type of a subcomponent, the component type definition has to exist in the denoted package. For example, the subcomponent declaration shown in Listing 3.38 uses the qualified type `ma.msg.Filter` (cf. l. 2). Hence, a component definition `Filter` has to exist in package `ma.msg`.

```
1 component A_B_Filter {                                                    MA
2   component ma.msg.Filter('a') af;                                        ...
3 }
```

Listing 3.38: R3: Using qualified subcomponent types.

R4: Unqualified subcomponent types either exist in the current package or are imported using an import statement.

If an unqualified component type is used as the type of a subcomponent, it must either exist in the current package or it must be imported using an import statement. Subcomponent `af` given in Listing 3.39 uses the unqualified type `Filter` which is imported in l. 2. The type of subcomponent `cdf` (l. 6) is unqualified and not imported. Therefore a component type definition `C_D_Filter` has to exist in the current package `ma` given in l. 1.

```
1 package ma;                                                    MA
2 import ma.msg.Filter;                                          ...
3
4 component A_B_Filter {
5   component Filter('a') af;
6   component C_D_Filter cdf;
7 }
```

Listing 3.39: R4: Using unqualified but imported subcomponent types.

R5: The first part of a qualified connector's source respectively target must correspond to a subcomponent declared in the current component definition.

If a source or target of a connector is qualified, the qualifier must be the name of a subcomponent that is declared in the namespace of the current component definition. In Listing 3.40, the target of the first connector (l. 7) is qualified with `af`. Since a subcomponent `af` is declared in l. 5, the qualifier is valid. In contrast, the source of the second connector (l. 8) is qualified with `bf`. Since a subcomponent with that name does not exist, this connector is invalid.

```
1 component A_B_Filter {                                         MA
2   port
3     in String msgIn,
4     out String msgOut;
5   component Filter('a') af;
6
7   connect msgIn -> af.msgs;
8   connect bf.filteredMsgs -> msgOut; ⊗ // subcomponent bf
9 }                                      // does not exist
```

Listing 3.40: R5: Subcomponents in qualified connector parts.

R6: The second part of a qualified connector's source respectively target must correspond to a port name of the referenced subcomponent determined by the first part.

The second part of a qualified source or target of a connector is a port name. A port with that name must exist in the component type of the subcomponent that is given by the qualifier. In

Listing 3.41, the target of the first connector given in l. 10 is port `toDelay` of subcomponent `del`. As shown in l. 7, the component type of this subcomponent contains this port. Hence, the first connector is valid. The source of the second connector (l. 11) is port `delayed` of subcomponent `del`. Since this port does not exist in component type `Delay` (cf. ll. 5–9), this connector is invalid.

```
1  component FilterDelay {                                          MA
2    port
3      in String inputData,
4      out String delayed;
5    component Delay del {
6      port
7        in String toDelay,
8        out String delayedData;
9    }
10   connect inputData -> del.toDelay;
11   connect del.delayed -> delayed; // out port delayed does not
12 }                                  // exist in type Delay
```

Listing 3.41: R6: Ports in qualified connector parts.

R7: The source port of a simple connector must exist in the subcomponents type.

In simple connectors, the source directly references an outgoing port of the type of the corresponding subcomponent. This port has to exist. In Listing 3.42, the source of the first simple connector in l. 11 exists and the connector is therefore valid. Since the component type `Delay` does not have an outgoing port `delayed` (cf. ll. 5–9), the second simple connector given in l. 12 is invalid.

```
1  component FilterDelay {                                          MA
2    port
3      out String delayed1,
4      out String delayed2;
5    component Delay {
6      port
7        in String toDelay,
8        out String delayedData;
9    }
10   component Delay
11     d1 [delayedData -> delayed1],
12     d2 [delayed -> delayed2]; // out port delayed does not
13 }                               // exist in type Delay
```

Listing 3.42: R7: Sources of simple connectors.

R8: The target port in a connection has to be compatible to the source port, i.e., the type of the target port is identical or a supertype of the source port type.

To assure type correct communication, source and target ports of connectors have to be compatible. A receiver can be connected to a sender if both have the same type or the receiver type is a supertype of the source type. Listing 3.43 contains some examples for connectors with different source and target types. The first connector in l. 9 is obviously valid since source and target type are the same. The second connector in l. 10 connects a source port with type `Integer` and a target port with type `Object`. Since `Object` is a supertype of `Integer`, this connection is valid. The third connector (l. 11) connects `Object` with `String`. Because `String` is a subtype of `Object` and not a supertype, it is invalid. The fourth connector in l. 12 is valid again because `Object` is a supertype of `String`.

```
1  component MyComp {                                                    MA
2    port
3      in Integer myInt,
4      out Object myObj;
5    component Buffer<Integer> bInt;
6    component Buffer<Object> bObj;
7    component Buffer<String> bStr;
8
9    connect myInt -> bInt.input;
10   connect bInt.buffered -> bObj.input;
11   connect bObj.buffered -> bStr.input; ⊗ // incompatible
12   connect bStr.buffered -> myObj;        // types Object and
13 }                                        // String
14 component Buffer<T> {
15   port
16     in T input,
17     out T buffered;
18 }
```

Listing 3.43: R8: Type compatible connectors.

R9: If a generic component type is instantiated as a subcomponent, all generic parameters have to be assigned.

A generic component is a component which defines generic type parameters in its head (see Section 3.2.1). If such a component type is used as a subcomponent type, a data type has to be assigned to each of these generic type parameters. Listing 3.44 contains the definition of the generic component type `Buffer` in ll. 1–5 which has two generic type parameters `K` and `V`. In the component type definition `MyComp` in ll. 6–9, two subcomponents are declared which have the aforementioned type. The first subcomponent declaration (l. 7.) assigns a data type to each type parameter and is valid. The incoming port `input` of `b1` has now the type `Integer`, the outgoing port has the type `String`. The second subcomponent declaration `b2`

in l. 8 only assigns one type parameter. Since the `Buffer` component type claims two generic type parameters, the subcomponent declaration is invalid.

```
1 component Buffer<K, V> {                                          MA
2   port                                                            ...
3     in K input,
4     out V buffered;
5 }
6 component MyComp {
7   component Buffer<Integer, String> b1;
8   component Buffer<Integer> b2;  // type parameter V
9 }                                // not assigned
```

Listing 3.44: R9: Using generic component types as subcomponent types.

R10: If a configurable component is instantiated as a subcomponent, all configuration parameters have to be assigned.

A configurable component defines configuration parameters in its head (see Section 3.2.1). If such a component type is used as a subcomponent type, a value has to be assigned to each configuration parameter. In Listing 3.45 the configurable component type `LossyDelay` defined in ll. 1–5 is used as type of subcomponent `ld1` in l. 7. In the subcomponent declaration, a value is assigned to both configuration parameters. Therefore, the subcomponent declaration is valid. The second subcomponent declaration in l. 8 only assigns one value. Since two values are expected, the declaration is invalid. Beside the amount of configuration parameters, the assigned values have to be type compatible with its matching configuration parameter.

```
1 component LossyDelay<T>[int delay, int lossrate] {                MA
2   port                                                            ...
3     in T msgIn,
4     out T delayed;
5 }
6 component MyComp {
7   component LossyDelay<String>(1, 5) ld1;
8   component LossyDelay<String>(1) ld2;  // missing parameter
9 }                                       // lossrate
```

Listing 3.45: R10: Using configurable component types as subcomponent types.

R11: Inheritance cycles of component types are forbidden.

Listing 3.46 shows an example for an inheritance cycle. The component type `ABPReceiver` extends the `CommonReceiver` component type (ll. 1f) which is a subtype of the `ABPReceiver` component (ll. 4f). Such a system cannot be instantiated. Consequently, inheritance cycles are forbidden.

```
1 component ABPReceiver<T>                                          MA
2     extends CommonReceiver<T> { // inheritance cycle              ...
3 }
4 component CommonReceiver<T>
5     extends ABPReceiver<T> { // inheritance cycle
6 }
```

Listing 3.46: R11: An inheritance cycle of components `ABPReceiver` and `CommonReceiver`.

R12: An inner component type definition must not extend the component type in which it is defined.

A structural extension cycle is given if an inner component type definition extends the component type of its surrounding parent component. Since the inner component will import itself in a structural extension cycle, it cannot be instantiated. Therefore, it is forbidden for inner component type definitions to extend its parent component. This context condition is violated in Listing 3.47. The inner component type `Inner` extends its surrounding component type `Outer` (l. 2).

```
1 component Outer {                                                 MA
2   component Inner extends Outer { // structural                   ...
3   }                                 // inheritance cycle
4 }
```

Listing 3.47: R12: Structural extension cycle.

R13: Subcomponent reference cycles are forbidden.

A reference cycle is given if two component types declare each other as subcomponets. Since instantiation of such a system will result in an endless instantiation process, these cycles are forbidden. An example of a reference cycle is depicted in Listing 3.48. Component type `A` contains a subcomponent declaration of type `B` (cf. l. 2). The component type `B` contains itself a subcomponent of type `A` (cf. l. 5). If component type `A` is instantiated, an instance of component type `B` is created, that will itself create another instance of `A` and so forth.

```
1 component A {                                                     MA
2   component B myB; // reference cycle
3 }
4 component B {
5   component A myA; // reference cycle
6 }
```

Listing 3.48: R13: Subcomponent reference cycle.

R14: Components that inherit from a parametrized component provide configuration parameters with the same types, but are allowed to provide more parameters.

A component definition that inherits from a configurable component definition has to define at least the same amount of configuration parameters like the supercomponent does. In addition, the types of the configuration parameters from the inheriting component have to match the types of the configuration parameters from the supercomponent. However, the names of the parameters are allowed to differ.

Listing 3.49 demonstrates this using the example of the configurable component type `A` which is inherited by several other component types. Component `B` (l. 3) does not define any configuration parameters and is, therefore, invalid. `C` (l. 5) defines a parameter with type `int` named `a`. The parameter types from `C` and `A` match, the parameter names, however, do not matter. In contrast to this, component `D` (l. 7) defines a parameter with type `String` named `x`. Here, the names match, but the types do not match. Hence, component `C` is defined correctly but component `D` is not defined correctly. Since it is allowed to define additional parameters on top of the parameters imposed by the supercomponent, component `E` (l. 9) is defined correctly. However, parameters must only be added to the end of the parameter list and parameter positions cannot be switched. This restriction is needed because parameters are automatically matched to the parameters of the supercomponent in the order of their occurrence. For that reason, component definition `F` (l. 11) is invalid because parameter `x` with type `String` is added to the head of the parameter list and is therefore matched with parameter `x` from component `A`. This is not allowed since both types are not compatible.

```
1  component A[int x] {...}                                          MA
2                                                                    ...
3  component B extends A {...} // B has to define an int parameter,
4                              // too
5  component C[int a] extends A {...} // int parameter defined.
6
7  component D[String x] extends A {...} // invalid parameter type.
8
9  component E[int a, String x] extends A {...}// additional
10                                             // parameters allowed.
11 component F[String x, int a] extends A {...} // Parameter order
12                                                // injured.
```

Listing 3.49: R14: Inheritance of component parameters.

R15: Components that inherit from a generic component have to assign concrete type arguments to all generic type parameters.

A component definition that inherits from a generic component has to assign type arguments to all generic type parameters of the supercomponent. If the type parameters of the supercomponent are restricted with an upper bound, the used type has to be compatible with the restricted

type parameter.

Listing 3.50 demonstrates this using the example of generic component type `A` which defines the generic type parameters `K` and `V` (l. 1). `V`'s upper bound is set to `Number`. Hence, the type `Number` or any subtype can be assigned to generic parameter `V`. Component `A` is inherited by several other component types. Component `B` (l. 3) simply assigns concrete types to the generic type parameters. Since `Integer` is a subtype of `Number`, component `B` is defined correctly. Generic component `C` (l. 5) assigns its own type parameters to the type parameters of its supercomponent. Since `C`'s type parameters are restricted the same way, this component definition is valid, too. Also partial forwarding of type parameters is possible. Component `D` assigns its own type parameter `K` to the first type parameter of its supercomponent (l. 7). Additionally `Integer` is assigned to `A`'s second type parameter `V`. Component `E` (l. 10) does not assign any type parameter to its supercomponent. Thus, component `E` is not defined correctly. Since all type parameters have to be assigned to a generic supercomponent, component `F` (l. 13) is also not valid. It only assigns a single type parameter to its supercomponent. As stated above, assigned types have to be compatible to restricted generic type parameters of a supercomponent. For this reason, component `G` (l. 15) is also defective because its type parameter `Y`, which is assigned to type parameter `V` of component `A`, is not compatible to the upper bound `Number`.

```
1  component A<K, V extends Number> {...}                    MA
2  // Concrete types assigned.                                ...
3  component B extends A<String, Integer> {...}
4  // Forward type parameters.
5  component C<K, V extends Number> extends A<K, V> {...}
6  // Partially forward type parameters
7  component D<K> extends A<K, Integer> {...}
8  // Type parameters 'K, V extends Number'
9  // of the extended super component have to be set.
10 component E extends A {...}
11 // All type parameters 'K, V extends Number' of the
12 // extended super component have to be set.
13 component F<T> extends A<T> {...}
14 // Y is not compatible with the upper bound of V (Number).
15 component G<X, Y> extends A<X, Y> {...}
```

Listing 3.50: R15: Inheritance of generic type parameters.

3.5.4 Conventions

Context conditions within this group define modeling conventions for MontiArc. Coding conventions for general purpose languages (GPLs) are commonly used and offer the advantage of a uniform code representation. Since familiar code representations are much easier to interpret, code sharing and distributed work is simplified. This advantage also holds for textual modeling languages. Injuring the given conventions usually results in a warning.

CV1: Instance names start with a lower-case letter.

Names in the scope of component definitions should start with a lower case letter. This context condition affects names of ports, subcomponent declarations, configuration parameters, and constraints. Therefore, all names contained in the component definition depicted in Listing 3.51 obey this rule. Violating this context condition will result in a warning.

```
1 component Inverter<T> [Number delta] {                    MA
2   port                                                    ...
3     in Integer input,
4     out Integer inverted;
5   component Filter(delta) myFilter;
6
7   java inv isInverted: {
8   }
9 }
```

Listing 3.51: CV1 and CV2: Naming Conventions of MontiArc.

CV2: Types start with an upper-case letter.

Component types and generic type parameters should start with an upper case letter. Hence, the component name `Inverter`, which is also depicted in Listing 3.51, as well as the used generic type parameter `T` are well formed. Violating this context condition will result in a warning.

CV3: Duplicated imports should be avoided.

Defining identical imports more than once will result in a warning.

CV4: Unused direct imports should be avoided.

The definition of imports which are not used within the model violates this convention and results in a warning.

CV5: In decomposed components, all ports should be used in at least one connector.

If incoming or outgoing ports of a decomposed component type are not used in at least one connector, a warning is produced to inform the modeler that parts of the components interface are unconnected. In Listing 3.52 the ports `msgIn` and `msgOut` are both used and connected to subcomponents (cf. ll. 8f). In contrast, port `foo` is not connected and a warning is raised (l. 4).

Please note that unconnected incoming ports and unconnected outgoing ports of subcomponents may result in a memory leak in the simulation. Since received or sent messages are buffered but never processed, the port buffer continuously grows. Hence, the modeler should connect these unused ports with a `Terminator` subcomponent which is available in the MontiArc model library (see Section 6.8.2). Alternatively, ports with restricted buffers can be used. A detailed description of the simulation and scheduling process is given in Chapter 4.

```
1 component A_Filter {                                          MA
2   port
3     in String msgIn,
4     in String foo,  // unused port
5     out String msgOut;
6   component Filter('a') af;
7
8   connect msgIn -> af.msgs;
9   connect af.filteredMsgs -> msgOut;
10 }
```

Listing 3.52: CV5: Using all ports.

CV6: All ports of subcomponents should be used in at least one connector.

If ports of subcomponents are unconnected, this can result in an unexpected behavior. Hence, the modeler is informed with a warning if subcomponents in a decomposed component type definition have unconnected ports. All ports of subcomponent `af` in Listing 3.53 are connected by the connectors given in ll. 8f. Since no ports of subcomponent `bf` are connected, a warning is raised.

```
1 component A_Filter {                                          MA
2   port
3     in String msgIn,
4     out String msgOut;
5   component Filter('a') af;
6   component Filter('b') bf;  // unconnected ports msgs and
7                              // filteredMsgs
8   connect msgIn -> af.msgs;
9   connect af.filteredMsgs -> msgOut;
10 }
```

Listing 3.53: CV6: Using all ports of subcomponents.

CV7: Avoid using implicit and explicit names for elements with the same type.

Implicit names for ports and subcomponents are derived from their unique type (see requirement *LRQ1.4*). A unique name is still given if an implicit name is used for an element of a certain type and further elements of the same type are explicitly named. To avoid confusion, implicit and explicit names for elements of a certain type should not be mixed. A warning is raised if both, implicit and explicit names, are used.

This is demonstrated in Listing 3.54. The port type `String` is used for three ports `msgIn`, `msgOut`, and `string` (ll. 3–5). Since the last port has no explicit name, a warning is raised. This convention is also injured by component type `Filter` which is used for subcomponent `af` (l. 7) and the implicitly named subcomponent `filter` (l. 8). Again, a warning is raised.

```
component A_Filter {                                                    MA
  port
    in String msgIn,
    out String msgOut,
    out String; // Implicit naming should be used
                // for unique port types only.
  component Filter('a') af;
  component Filter('b'); // Implicit naming should be used
                         // for unique subcomponent types only.
  connect msgIn -> af.msgs, filter.msgs;
  connect af.filteredMsgs -> msgOut;
  connect filter.filteredMsgs -> string;
}
```

Listing 3.54: CV7: Using implicit and explicit names for elements with the same type.

3.5.5 Code Generation

Context conditions within this category validate certain properties of a model to ensure that the generated code can be executed within a MontiArc simulation.

CG1: Communication cycles without delay should be avoided.

Communication cycles are created if a subcomponent of a decomposed component directly or indirectly communicates with itself. Due to the used simulation time paradigm taken from FOCUS and the realized scheduling, ticks are used to synchronize timing at incoming ports of a component (see scheduling of ticks in Section 4.3.2 on page 100). A tick is a special event used to model the borders of time intervals within a stream. If all incoming ports of a component received a tick, the component will emit a tick on all outgoing ports. This leads to the Brock-Ackerman anomaly [BA81]. In MontiArc this anomaly is present if a subcomponent, which directly or indirectly communicates with itself, waits for ticks on incoming ports, that it has to emit itself. This, however, will result in a simulation deadlock. Such a simulation deadlock can be broken if a delay is introduced into a communication cycle at initialization time of the simulation.

A delay is produced by explicit delay components (see Section 6.8.2) or components that belong to the delayed or causal synchronous time domain (cf. Section 4.4). Such components initially emit at least one tick on their outgoing ports. In this way, they have a simulated delay and can provide initial ticks on incoming ports of a communication cycle. This context condition analyses decomposed components and finds communication cycles. This is done by step-wise creating a directed graph of connected non-delaying subcomponents which is analysed for cycles.

An instance of a communication cycle is given in Figure 3.55. The decomposed component `ContAdder` contains a subcomponent named `adder` whose output emitted by port `c` is transmitted to subcomponent `initZero`. The latter is connected to the incoming port `b` of the `adder` subcomponent. If at least one component `InitZero` or `Adder` belongs to the delayed

or causal synchronous timing domain, this cycle does not cause a simulation deadlock. Else a warning is raised. Please note that only a warning and not an error is raised because the resulting deadlock only affects the MontiArc simulation and not the system under development (SUD). Furthermore, this deadlock analysis is only performed on MontiArc models and not on the implementation. If delayed or causal synchronous components in a feedback cycle are not labeled correctly, the detected cycle will not result in a simulation deadlock, too.

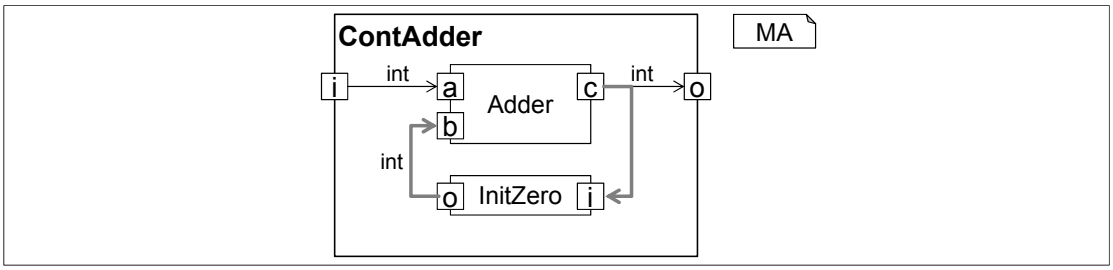


Figure 3.55: CG1: Communication cycle between subcomponents.

3.6 AADL Compatibility

An industrial survey discussed in [MLM+13], based on interviews of 48 practitioners from 40 different IT companies, revealed that the most commonly used ADLs are the Unified Modeling Language (UML), the AADL, and ArchiMate. The UML contains several diagrams that in combination are suitable to define architectures. Component diagrams, composite structure diagrams, as well as deployment diagrams are provided. Nevertheless, the UML with its graphical notation is very heavyweight and the concrete semantics of the named diagram kinds is tool respectively user dependent (see, e.g., [OL06]). Thus, additional interpretations are developed and implemented in tools which are not always consistent with the intentions of the original UML diagrams. ArchiMate [LAPJ10] aims at modeling enterprise system architectures and thus does not really match the attended architectural style. Consequently, it has been decided to ensure compatibility to AADL models (see requirement *LRQ1.8*).

The main properties of AADL [FGH06, FG12, SAE12] have already been introduced in Section 2.3.1. It is mostly used to model static architectures of embedded systems, but also provides a restricted form of dynamics with the definition of modes. In the following, MontiArc's modeling elements presented in Section 3.2 are mapped to equivalent AADL modeling elements. Please note, an excerpt of the meta-model which defines the AADL language and a complete listings with the used AADL examples are given in Appendix D.

3.6.1 AADL Components

AADL components are types which are defined in a package and belong to a certain category. In contrast to MontiArc, where a single reusable component is defined in a compilation unit, the AADL compilation unit is given by a package which contains multiple component definitions. Nevertheless, it is also possible to define multiple inner components within a MontiArc

component. But, in contrast to AADL packages where each contained component can be instantiated as a subcomponent, inner MontiArc components can only be used within the scope of the surrounding component type. Thus, they can be used to divide the implementation of a reusable component type into distinct inner components which are encapsulated and hidden from the environment.

AADL software component categories comprise threads, thread groups, processes, data, as well as subprograms of the modeled system. Execution platform respectively hardware components represent processors, memory, devices, or data buses. A composite component represents a system and its decomposition. Generic or abstract components are not yet assigned to a category. Thus, AADL diagrams include both, the software and hardware architecture. The mapping from the former to the latter is defined by hierarchical containment. Since MontiArc focuses on logical architectures only, an atomic MontiArc component can represent both: a software or hardware component. Hence, it corresponds to an abstract AADL component and a decomposed MontiArc component conforms to an AADL system component.

The AADL language distinguishes between a *component type* and a *component implementation*. The former contains features that describe the externally visible interface which comprises ports, visible attributes, and properties. Further, a component type can extend another component type and it can contain prototypes that are used to parametrize interfaces. Flows define input-output port dependencies of a type and modes describe different operation modes. Component implementations have to implement a component type and inherit its features. Additionally, an implementation can extend another component implementation to inherit and refine its internal structure. Implementations can contain subcomponents, connections, subprogram calls, modes and transitions between modes, flow implementations, which describe data flow across a sequence of subcomponents, and further not externally visible properties.

On the one hand, the distinction between type and implementation definition has the benefit that different implementations of the same interface can be created and exchanged (polymorphism). On the other hand, complex AADL implementations are harder to understand since their port interface is not included in the implementation definition but elements of the interface, e.g., ports, have to be referenced. Since all component implementations have to be kept in a consistent state when a component type is changed, the evolution of these separated artifacts is complicated. If a strict approach of defining a separate interface for each component type is to be used, atomic MontiArc components can be used to emulate component interfaces. Thus, both forms of representation can be mapped into each other, but MontiArc provides more comfort and compacter design.

AADL

```
1 abstract A
2 -- ...
3 end A;
4 abstract implementation A.AImpl
5 -- ...
6 end A.AImpl;
```

Listing 3.56: AADL specification of component type `A` and its implementation `AImpl`.

Listing 3.56 depicts the specification of component type `A` (ll. 1 – 3) and its component implementation `AImpl` (ll. 4 – 6) which correspond to the atomic MontiArc component depicted in Listing 3.2. Component type specifications start with the category of the component (`abstract`) followed by the type name. Component implementations repeat the category and reference the implemented type after the keyword `implementation`. The implementation name is appended to the implemented type separated with a dot. Thus, component implementation `A.AImpl` implements component type `A` and is named `AImpl`.

AADL supports component type and implementation inheritance. Basically, if one type or implementation extends another type or implementation, all model elements are inherited and can be refined. The specification of AADL component type `Ext` and its implementation `ExtImpl` given in Listing 3.57 correspond to the MontiArc component `Ext` depicted in Listing 3.3. The listing demonstrates how to use inheritance in AADL. The component type (ll. 1f) extends another component type which has to belong to the same category (`abstract`). The component implementation can also extend another component implementation which implements the same component type or a subtype. Consequently, the type hierarchy of component implementations has to be aligned with the type hierarchy of the implemented component types.

```
1  abstract Ext extends A                                    AADL
2  end Ext;
3  abstract implementation Ext.ExtImpl extends A.AImpl
4  end Ext.ExtImpl;
```

Listing 3.57: AADL specification of component type `Ext` and its implementation `ExtImpl` as an extension of component type `A` and implementation `AImpl`.

AADL does not directly support the definition of *configurable components* with configuration parameters. But these can be either emulated using properties defined in a global property set or with data ports which model shared data. Since the values of data ports may vary at runtime, properties are more suitable to model the values of configuration parameters which are assigned at instantiation time once. The Listings 3.58 and 3.59 exemplary depict how to emulate configuration parameters in AADL. The property set definition given in the former listing defines the configuration parameters `p1` and `p2` for component type `B` which is defined in package `Snippets` (ll. 4f). Optionally, default values can be assigned to these parameters in the definition of the component type (Listing 3.59, ll. 3f). Along with the defined property set, this component type is an equivalent to the MontiArc component given in Listing 3.4.

Since AADL 2.0, the language provides prototypes that can be used to define *generic component types* or generic component implementations. MontiArc's type parameters can be directly translated to AADL prototypes with the category `data`. AADL component specification `C`, which is an equivalent to the MontiArc component given in Listing 3.5, demonstrates the use of data prototypes to define the generic type parameters `K` and `V`. (Listing 3.60).

```
property set Snippets_cfg is
  with Snippets;

  p1: aadlinteger applies to (Snippets::B);
  p2: aadlstring applies to (Snippets::B);

end Snippets_cfg;
```

Listing 3.58: AADL property set which defines the configuration parameters of a configurable component.

```
  abstract B
    properties
      Snippets_cfg::p1=>1;
      Snippets_cfg::p2=>"bar";
  end B;
```

Listing 3.59: Default values of configuration parameters in AADL.

```
  abstract C
    prototypes
      K: data;
      V: data Base_Types::Natural;
  end C;
```

Listing 3.60: Defining type parameters with AADL data prototypes.

3.6.2 AADL Interfaces

AADL provides several interface model elements called features. These can be used within AADL component types (see above) and are listed in the following. Feature groups allow to define reusable composites of features. Data, event data, and event ports are used to model unqueued, queued, and asynchronous event communication. Additionally, required or provided access to data, buses, or subprograms can be part of a component type. Finally, parameters can be defined, that are, e.g., forwarded to subprograms. MontiArc ports correspond to AADL event data ports which are restricted to a single direction (in or out). Listing 3.61 depicts the interface definition of component `A` that is equivalent to the MontiArc component given in Listing 3.6. The feature list is introduced by the keyword `features`, followed by a list of incoming and outgoing event data ports. A port definition starts with the port name, followed by the direction (`in` or `out`), its kind (`event data port`), and its data type (`String, Cmd, Integer`).

AADL provides its own type system which can be used to define port data types. Osate2 offers a set of predefined data components, such as `String` or `Integer`, that are defined in the package `Base_Types`. If these types are referenced, e.g., in a port definition, they have to be qualified with the package name (see Listing 3.61, ll. 3, 5). To convert a MontiArc component to AADL, the used types have to be transformed to data components that define structured data

```
abstract A                                                          AADL
  features
    string: in event data port Base_Types::String;
    command: in event data port Cmd;
    integer: out event data port Base_Types::Integer;
end A;
```

Listing 3.61: AADL interface definition of component type A.

in a composite way. The data component Cmd, which defines the data type of port command, is depicted in Listing 3.62. The allowed values of the enumeration are defined using properties from the Data_Model package. The type corresponds to the enumeration defined in the UML/P CD given in Listing 3.7.

```
data Cmd                                                            AADL
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators=>("PULL", "PUSH");
    Data_Model::Representation => ("0", "1");
end Cmd;
```

Listing 3.62: An enumerated data component that defines data type Cmd.

Interfaces of generic components can use the defined data prototypes as port data types. The features of generic component type C (see Listing 3.60) are depicted in Listing 3.63. The port msgIn has prototype V as data type, prototype K is used for port msgOut.

```
  features                                                          AADL
    msgIn: in event data port V;
    msgOut: out event data port K;
```

Listing 3.63: Using prototypes as port data types in AADL.

3.6.3 AADL Architectural Configuration

The architectural configuration of an AADL component is given by a component implementation which contains subcomponents and connections. A component implementation can further contain annexes, modes, and flow implementations. The semantic of a MontiArc component instance directly corresponds to the semantic of an AADL component instance. Therefore, a component instance represents the instantiation of a component type definition and the following recursive instantiation of contained subcomponents. In AADL, a *subcomponent* instantiates a component implementation and associates a local name. MontiArc subcomponents directly correspond to AADL subcomponents regarding the constraint, that the AADL component category of the subcomponent is restricted to abstract or system.

Beside port connections, AADL provides component access connections, e.g., to model shared data, subprogram calls to control the sequence of called subprograms, and parameter connec-

tions, e.g., to provide the return value of a subprogram call via an outgoing port. Since shared memory is forbidden in MontiArc (see requirement *LRQ1.1.5*) and subprogram calls are not available, similar connectors are not included in the MontiArc language. AADL provides several connector kinds with distinct communication rules. The transmission kind of port connections is determined by the connected port kinds (delayed, queueing, allowed amount of senders and receivers). Since MontiArc only provides a single port kind, MontiArc connectors correspond to AADL port connections with immediate and queued transmission which connect one sender with an arbitrary amount of receivers (see requirement *LRQ1.3*).

An AADL example that demonstrates how to instantiate components as subcomponents and how to create connections is given in Listing 3.64. The listing depicts an equivalent AADL version of the MontiArc component that is shown in Listing 3.9 and 3.10. The `subcomponents` clause contains the subcomponents which are instantiated by the system implementation `DImpl` (ll. 9 – 21). Subcomponent `a` is declared in l. 10. A subcomponent declaration starts with the name (`a`), followed by the category (`abstract`), and the instantiated component implementation `A.AImpl`. The configuration values of configurable components are set within curly brackets (ll. 13 – 16). Thus, a `5` is assigned to parameter `p1` and the String `"foo"` is assigned to parameter `p2` of subcomponent `myB1`. The previously defined default configuration values (see Listing 3.59) are overridden when concrete values are assigned to a configurable subcomponent. A generic subcomponent is instantiated in ll. 18 – 21. Concrete data components (`String` and `Integer`) are assigned to the type parameters `K` and `V` within round brackets. Thus, port `msgIn` has the data type `Integer` and port `msgOut` the data type `String` (see Listing 3.63).

The ports that are declared in the component type (ll. 1 – 6) are connected to the ports of the subcomponents in the `connections` clause (ll. 23 – 28). An AADL connection always has a name (`cona`, `conb`, etc.) followed by the connection kind (`port`). A single source is connected with a single target. For example, connector `cona` (cf. l. 24) connects the port `sIn`, which is defined in the component type (l. 3), with port `sIn` of subcomponent `myB1`.

In contrast to MontiArc, AADL does not directly support the definition of *inner components*. However, this concept can be partially emulated by encapsulating the defining component together with the inner component alone into a package and restrict the visibility of the inner component implementation to private. Since AADL is not able to reference private types within the public section of the package, the type of the inner component still has to be public. An example for this method is depicted in Listing 3.65. It emulates the definition of the inner MontiArc component `InnerA` which is given in Listing 3.11. AADL packages are divided into a public and a private section. The latter contains the implementation of the abstract component type `InnerA.InnerAImpl` (ll. 13 – 14). The former defines a system component type `F` (ll. 2f), the component type of the inner component `InnerA` (ll. 4f), and the implementation of the system component `F.FImpl` (ll. 7 – 10). The system implementation instantiates the private component type `InnerA` as a subcomponent `innerA` (l. 9). In this way, the inner component implementaton `InnerA.InnerAImpl` cannot be used outside the current package. However, it cannot be used in the private section of the depicted package as well.

```
AADL
system D
  features
    sIn: in event data port Base_Types::String;
    sOut: out event data port Base_Types::String;
    iOut: out event data port Base_Types::Integer;
end D;

system implementation D.DImpl
  subcomponents
    a: abstract A.AImpl;
    myExt: abstract Ext.ExtImpl;

    myB1: abstract B.BImpl {
      Snippets_cfg::p1=>5;
      Snippets_cfg::p2=>"foo";
    };

    c: abstract C.CImpl (
      K => data Base_Types::String,
      V => data Base_Types::Integer
    );

  connections
    cona: port sIn -> myB1.sIn;
    conb: port c.msgOut -> sOut;
    conc: port myB1.sOut -> a.string;
    cond: port myB1.sOut -> myExt.string;
    cone: port myExt.integer -> iOut;
end D.DImpl;
```

Listing 3.64: Subcomponents and connections in AADL.

3.6.4 Further AADL Modeling Elements

As already mentioned, AADL is a rich language which further provides modeling elements that are not or not directly supported by MontiArc. These are briefly introduced in the following.

Annexes serve as extensions of the AADL language to integrate languages designed for a special purpose. For instance, the Behavior Annex [SAE11] adds abstract state machines into component implementations which can be used to directly implement the behavior of components. Language elements provided by an AADL annex can be integrated into the MontiArc language with a language extension (see Chapter 7). Therefore, MontiArc provides an extension point that allows to directly embed behavior implementations as architectural elements into components (see requirement *LRQ3.2*). This extension point is not used by MontiArc itself, but existing extensions like AJava [HRR10] (see Section 9.2) or MontiArcAutomaton [RRW13b, RRW13a, RRW13c] (see Section 9.3) implement this extension point by adding Java respectively I/O$^{\omega}$ automata [Rum96] into MontiArc components.

AADL *modes* are given by mode automata that are triggered by events, e.g., received mes-

```
AADL
public
  system F
  end F;
  abstract InnerA
  end InnerA;

  system implementation F.FImpl
    subcomponents
      innerA: abstract InnerA;
  end F.FImpl;

private
  abstract implementation InnerA.InnerAImpl
  end InnerA.InnerAImpl;
```

Listing 3.65: Emulating inner component definitions in AADL.

sages. A mode automata defines different states of a component. By mapping component elements to certain states, the configuration and behavior of a component can be switched by changing its state. Mode dynamics can be simulated in MontiArc by using delta modeling techniques [CHS10, HRRS11, HKR+11] at runtime or dedicated components that control the data flow and redirect it to subsystems that represent a certain state.

Flows and *Flow implementations* are used to define the information flow of messages through a component as well as the relation of input and output features. These information are, for example, used to calculate the execution time of a component. Flows can be simulated in MontiArc by annotating elements with stereotypes that document the relation between different ports and subcomponents.

3.6.5 Summary

An overview of the comparison of the basic modeling elements of MontiArc and AADL is given in Table 3.66. Further, Table 3.67 compares advanced language concepts of MontiArc and AADL.

Language Concept	MontiArc	AADL
Compilation unit	Component definition	Package with component types and implementations
Component categories	None (logical components)	Software, execution platform, composite, and generic/abstract components
Atomic component	Component definition w/o subcomponents	Software, execution platform, and abstract component type and implementation

Table 3.66 continued on next page

Language Concept	MontiArc	AADL
Decomposed component	Component definition with subcomponents	Composite component type and implementation
Inheritance	Component definition	Component type & Component implementation
Configurable components	Configuration parameters	Emulated: property set & properties
Generic components	Type parameters	Data prototypes
Component interface	Component definition with ports	Component type with features
Interaction points	Unidirectional data event ports	Features are: feature groups, uni- or bidirectional data, event, and data event ports; required/provided access to data, bus, or subprograms; parameters passed to subprogram calls
Data types	External: CD and Java	Internal: data component types and implementations
Architectural configuration	Decomposed component definition with subcomponents and connectors	Composite component implementation with subcomponents, immediate or delayed connections, and advanced modeling elements (see Table 3.67)
Atomic subcomponent	Subcomponent	Software, execution platform, or abstract subcomponent
Decomposed subcomponent	Subcomponent	System subcomponent
Channel	Undelayed connector	Uni/bidirectional, (un-)delayed port connection, data access connections, parameter passing, subprogram calls
Local component definition	Inner component definition	Partially emulated: private package

Table 3.66: Overview of basic language concepts and their representation in MontiArc and AADL.

Language Concept	MontiArc	AADL
Component timing	X	–
Autoconnect	X	–
Autoinstantiate	X	–
Constraints	OCL & Java	Constraint Annex [HG13]
Grouping of elements	–	feature & property groups
Message flows	Emulated: Stereotypes	Flows & flow implementations
Modes	Emulated: Δ-MontiArc or mode-switch component & subcomponent for each mode	modes & mode automata
Properties	Emulated: Stereotypes	Predefined and user-defined properties
Extension points	Embedding for behavior implementation and constraints, language inheritance for arbitrary extensions, extension method	Annex mechanism

Table 3.67: Overview of advanced language concepts and their availability in MontiArc and AADL.

Chapter 4

Simulating MontiArc Models

In Chapter 3, MontiArc, a simple and compact architecture description language (ADL), which is intended as a front end for agile architectural prototyping is presented. To execute and validate developed MontiArc components, a timed simulation of MontiArc models is helpful. This chapter presents the developed MontiArc simulation which has the following core features:

- Asynchronous, event-based communication between components of an interactive system is simulated using FOCUS [BDD+93, BS01] timed streams.
- Timed behavior of components can be simulated. The simulation time is completely independent from the real-time of the simulator. Thus, simulations can be executed much faster than real-time.
- Different component timing domains can be combined with each other.
- The simulation is deterministic to allow for deterministic component black-box and white-box tests. For this purpose, non-deterministic component implementations can be replaced with mock components.
- The simulation can be flexibly extended or adjusted by, e.g., using a custom scheduler or custom ports.
- Since the simulation is developed in Java, it can be executed on every platform with an available Java Virtual Machine (JVM).

Please note that no visualisation of the simulated communication is provided. The simulation is rather executed without user interaction to continuously and automatically test MontiArc components.

The chapter is organized as follows. After introducing the FOCUS foundations, the structure of the simulation runtime environment (RTE) is stepwise derived from the intended runtime structure in Section 4.2. The simulation scheduler is presented in detail by discussing all relevant scheduling scenarios in Section 4.3. Component behavior classes are defined in Section 4.4. These allow to realize and combine MontiArc components with distinct timed behavior. As claimed by requirement *SRQ10*, further optimizations of the presented scheduler strategy are discussed in Section 4.5. Finally, technical design decisions made during the development of the presented MontiArc simulation are concluded in Section 4.6.

4.1 Foundations for the Simulator

In practice, there exist various paradigms to model software architectures. Regarding these paradigms, that also describe how components communicate, interact, and behave, different ap-

proaches are suitable to simulate a system architecture. According to ter Beek et. al [tBFGM08], systems are most often described either *state-based* or *event-based*.

In the state-based paradigm, systems are described using states and transitions between states. Components are modeled using state machines like UML statecharts (SCs) [OMG11b] that mostly communicate via shared variables, as for example in [CCO+05]. According to Hoare [Hoa85], this leads to unwanted interferences and non-deterministic behavior. These problems can be solved by a controlled access to these shared variables, but this results in overhead and leads to deadlocks if a component does not release a used variable.

Event-based systems are characterized using events and actions. In FOCUS [BS01], component behavior is described using input-output patterns or recursive stream-processing functions which define reactions to specific input events. Also state-based behavior definitions are included in FOCUS. In contrast to the state-based paradigm, state is encapsulated in components [Rum96]. Event-based communication is characterized by asynchronous message passing between components.

Fraikin et al. [FFL05] compare these two paradigms on the example of the formal method B [Abr96] (state-based) and EB^3 [FSD03] (event-based), which are both used to model information systems. The result of this comparison is, that the event-based method is more related to the abstract user-scenario than the state-based method. And, in contrast, state-based models are more similar to the resulting program implementation. Also specifications of order constraints, such as assertions over the order of events, are more complex and harder to understand in a state-based paradigm.

Summing up, the event-based paradigm is suited to model and simulate a distributed interactive system in which components usually do not share state and communicate asynchronously using messages. Since architectural descriptions abstract from the concrete implementation, state-based approaches are less adequate to model such systems. Consequently, the event-based paradigm has been selected.

Thus, the architecture simulation, which is presented in this thesis, is based on the event-based paradigm taken from the FOCUS framework [BDD+93, BS01] for the development and modeling of distributed interactive systems. FOCUS provides mathematical foundations to model timed communication and the behavior of components based on a higher-order two-valued logic. In the following, the relevant concepts of FOCUS taken from [BDD+93, BS01] are presented and summed-up. If needed, the notation suggested in [RR11] has been used for formalization.

Communication

A central concept of FOCUS are *Streams*, a finite or possibly infinite sequence of elements that belong to a certain domain M. According to Ringert and Rumpe [RR11], this domain can be an abstraction of:

- event signals like bus messages,
- continuous values like sensor values or discrete event signals,
- simple messages for method invocations or signaling, or
- complex data structures that are passed between software services.

The notation of a stream is M^* for a finite and M^∞ for an infinite sequence of messages. $M^\omega = M^* \cup M^\infty$ denotes finite and infinite streams. A sequence of actions is named *trace*. In accordance to Hoare [Hoa85], a trace describes the behavior of a process in the finite sequence of symbols. These symbols represent events which either stimulate the process or are raised by the process itself at a certain point in time. Traces are used to model the behavior of a complete (closed) system.

A stream of messages describes the communication history between components over a *directed channel*. A directed channel unidirectionally transmits messages between an outgoing and an incoming port. Transmission over a channel is instantaneous, which means that it does not consume time. This property is based on the assumption that message processing is slow, while message transmission is fast. Further, a channel is reliable and order preserving. This means that no messages are lost and messages do not pass each other during the transmission. If transmission properties, such as message loss or rearrangement, are needed, they have to be explicitly modeled.

Components of a system are connected to *information-flow architectures* via incoming and outgoing channels. A channel can transmit messages of a certain type solely. If two components are attended to interact with each other, they have to be connected using at least two contrary directed channels that build a feedback cycle. Regarding these properties, channels are well suited to represent MontiArc connectors in a simulation (cf. Section 3.4). The FOCUS framework offers different kinds of streams that are described in the following. In the following, a stream of messages is called stream while a stream of actions is called trace.

[RR11] distinguishes *event streams* (respectively untimed streams) and *timed event streams* (timed streams). An event stream is an ordered sequence of messages which are transmitted over a directed channel. All messages within a stream are elements of the stream's domain. Hence, they have a compatible data type. A timed event stream $M^{\underline{\omega}} = (M \cup \{\surd\})^\omega$ additionally contains special events, so called ticks ($\surd \notin M$), which model the processing of time. Please note that an infinite observation of a channel is a timed stream with infinitely many ticks: $\forall s \in M^{\underline{\omega}} : \#_t s = \infty$. Nevertheless, since the executed simulations are finite, the contained streams are finite, too.

Ticks divide a timed stream into *time intervals*, where a time interval contains a finite amount of messages. All time intervals are logically equidistant. Within a time interval, the order of the contained messages is given but not the exact timing. For example, a system can be modeled which is capable to transmit a message each millisecond and the duration of one hour is chosen for a time interval. If such an interval contains three messages, they can be mapped to a certain hour, but these messages may be transmitted within the very first three milliseconds of this hour, the very last, or they may be equally distributed. However, the information about the order of these messages is given. Please note that due to this property, no assumptions about the order of messages transmitted in the same time interval over different channels can be made. This way FOCUS offers a discrete description of time within the time intervals of a timed stream. Since all time intervals logically represent the same fix period, time is continuously described as well. This time model is similar to the superdense time presented in [Pto14, Section 1.7].

Timed event streams can always be transformed into an untimed event stream by simply removing all ticks. However, an automatic transformation from untimed to timed streams is

Let $s, s' \in M^\omega, ts \in M^{\underline{\omega}}, n \in \mathbb{N}_\infty, a \in M, A \in (M \to \mathbb{B})$:

Notation	Signature	Description
$ft(s)$	$M^\omega \to M$	first: get first element of s
$s.n$	$\mathbb{N} \times M^\omega \to M$	nth: get n^{th} element of s
$rt(s)$	$M^\omega \to M^\omega$	rest: get stream without first element
$a : s$	$M \times M^\omega \to M^\omega$	append: appends a to the head of s
$s^\frown s'$	$M^\omega \times M^\omega \to M^\omega$	concat: concatenation of stream s and s'
$s \sqsubseteq s'$	$M^\omega \times M^\omega \to \mathbb{B}$	prefix: checks, if s is prefix of s'
$s\|_n$	$\mathbb{N}_\infty \times M^\omega \to M^\omega$	take: get prefix of s
$\#s$	$M^\omega \to \mathbb{N}_\infty$	length: get the length of stream s
$A\circledS s$	$(M \to \mathbb{B}) \times M^\omega \to M^\omega$	filter: filters away all elements for which the predicate function $(M \to \mathbb{B})$ not holds
$\#_t ts$	$M^{\underline{\omega}} \to \mathbb{N}_\infty$	time: number of time events ($\surd$s) in ts
$ts._t n$	$\mathbb{N} \times M^{\underline{\omega}} \to M^*$	tnth: get n^{th} time interval of ts
$ts \downarrow_n$	$\mathbb{N}_\infty \times M^{\underline{\omega}} \to M^{\underline{\omega}}$	ttake: get first n time intervals of ts
$\diamond ts$	$M^{\underline{\omega}} \to M^\omega$	abstraction: abstracts timed event stream ts to an event stream.

Table 4.1: FOCUS operators for untimed and timed streams based on [RR11].

only possible under certain condition since infinitely many timed representations of an untimed stream exist. One possible, but very specific timed interpretation of an untimed stream is given by *time-synchronous streams* which contain exactly one event for every point of time [RR11]. Thus, a time-synchronous stream is a timed stream, where exactly each second message event is a $\surd$. Mathematically, a time-synchronous stream is a function $\mathbb{N} \to M$ that maps the time interval index of a message to the message itself. A similar time-synchronous stream notation uses a pseudo message $\perp$ to model that no data has been transmitted within a time interval. Since every element in a stream denotes a new time interval, explicit modeling of ticks can be omitted. This stream kind extends the function of timed streams to $\mathbb{N} \to M_\perp$. If the logical duration in the example above is reduced from one hour to the transmission frequency of the system (a millisecond), then the system can transmit one message each time interval at max. This way, each message can be mapped to an exact discrete time interval and the communication is modeled time-synchronously.

If timed and untimed streams are compared, they syntactically differ because the former contain ticks. If a $\surd$ is interpreted as a regular message, all operations which are suitable for untimed streams can be also applied to timed streams. The fundamental difference between a $\surd$ and a message is the semantic interpretation in the underlying model. A $\surd$ in a stream represents the incrementation of a global clock [GGR06]. A message in a stream represents the transmission of this message over the corresponding directed channel. Thus, explicit time events in a simulator are the basis to decouple simulation and simulator time. Available stream operations are, e.g.,

defined in [BS01, RR11]. In Table 4.1, an excerpt is listed. The second part of the table contains special operations that are only available for timed streams $M^{\underline{\omega}}$. These operators allow stream manipulations and give information about existing streams.

4.2 Runtime Environment

MontiArc provides its own runtime environment (RTE) written in Java that realizes the communication foundations of FOCUS. It contains predefined Java interfaces and classes to develop components, their ports, and connections. Beside these infrastructural elements, the RTE also contains standard schedulers which handle message passing and the simulation of time. Classes generated by MontiArc are based on these RTE classes and interfaces. To provide a better understanding of the RTE, it is first explained how MontiArc components should be represented at runtime of a simulation. Then suitable classes of the RTE are derived.

4.2.1 Intended Object Structure @Runtime

The concrete representation of MontiArc components, ports, and subcomponents in the RTE and a running simulation is depicted in Table 4.2.

Element	**Realization in Java**
Component definition	Generated Java class that implements the RTE component interface
Port implementation	RTE class with generic message types
Stream implementation	RTE class with generic message types
Scheduler implementation	RTE class
Subcomponent or simulated system	Object of the generated Java class
Port	Object of the RTE port implementation

Table 4.2: Representation of MontiArc elements in the simulation.

It can be seen that the simulated system, subcomponents of this system, and ports are represented by objects during the simulation. The relations between these objects is explained by means of a concrete example given in Figure 4.3. It depicts an excerpt of component `LightCtrl`, which has already been presented in Section 3.1. The excerpt contains two subcomponents `DoorEval` and `Arbiter` and three different connector kinds. The first one (dashed line) connects an incoming port of a decomposed component with the incoming port of a subcomponent, the second one (gray) connects an outgoing port of a subcomponent with an incoming port of another subcomponent, and the third one (black) connects the outgoing port of a subcomponent with the outgoing port of the shown decomposed component. In the following, this diagram is stepwise instantiated as simulation objects in a straight forward way. Finally, the resulting object structure is optimized according to requirement *SRQ10*.

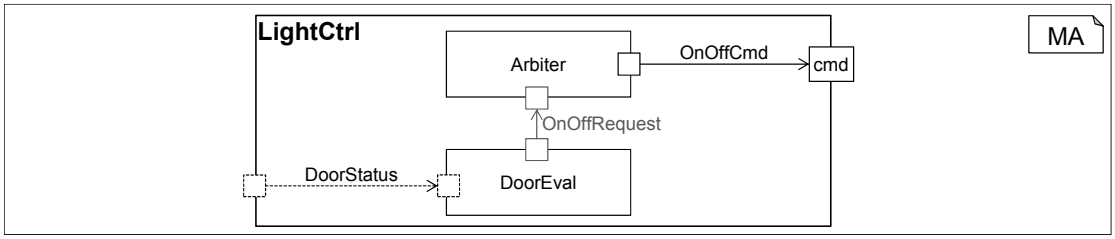


Figure 4.3: Excerpt of component LightCtrl.

Object Instantiation of a Simulation

Step 1: Components are instantiated from top to bottom. This means that system components are instantiated first, then their subcomponents, and then the subcomponents of these subcomponents and so forth. An example is depicted in Figure 4.4. As indicated by the arrow on the right-hand side, first a `LightCtrl` object is created, then an instance of each subcomponent's type is created and linked to the `LightCtrl` object.

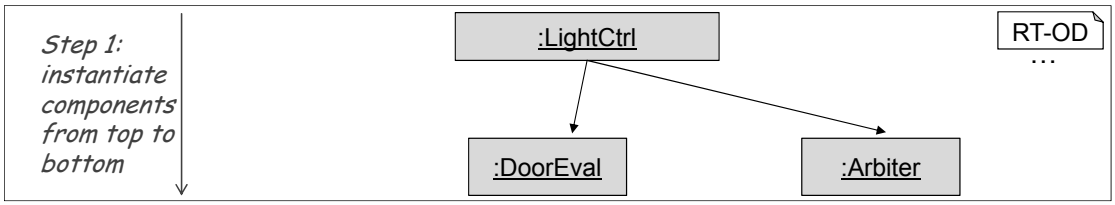


Figure 4.4: Instantiation of components as simulation objects. Step 1: instantiate components from top to bottom.

Step 2: Instantiate ports of atomic components. Hence, incoming and outgoing ports are created and linked to its corresponding component instance. Please note that the elements of the following runtime object diagrams (ODs), which represent the above mentioned connector kinds, have the same graphical style like the corresponding connection (cf. Figure 4.5).

Step 3: For each decomposed component from bottom to top:

Step 3.1: Create and link forwarding ports. This way port `doorStatus` is created in Figure 4.6. Please note that a forwarding port simply forwards received messages to the connected incoming ports of subcomponents.

Step 3.2: Create connections by linking ports. In Figure 4.7, the incoming forwarding port `doorStatus`, which belongs to component `LightCtrl`, is connected with the incoming port `doorStatus` of subcomponent `DoorEval`. The outgoing port of `DoorEval` is connected with the incoming port of subcomponent `Arbiter`. The connector, which connects `Arbiter`'s port `onOffCmd`

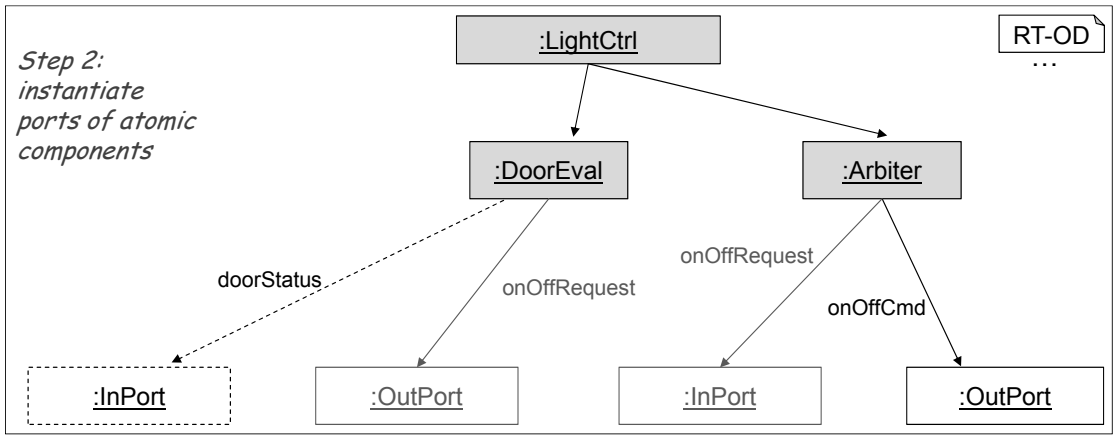


Figure 4.5: Instantiation of components as simulation objects. Step 2: instantiate ports of atomic components.

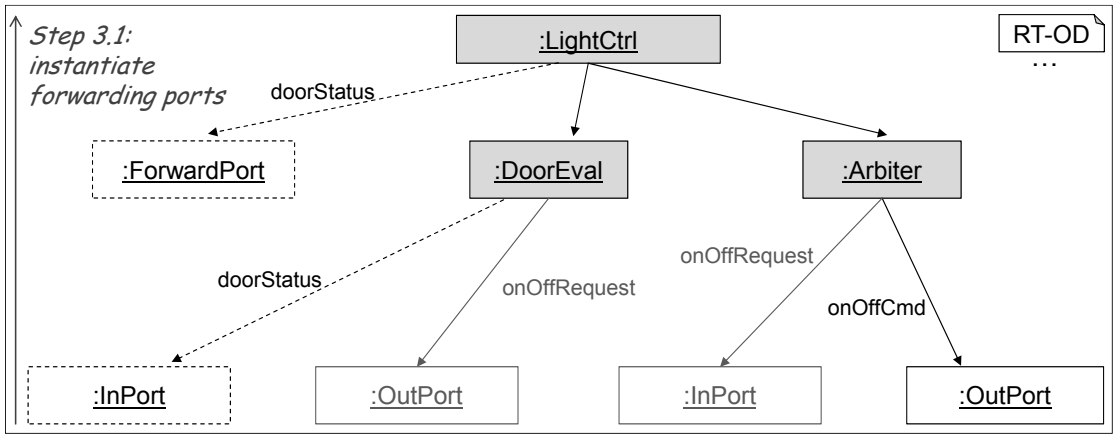


Figure 4.6: Instantiation of components as simulation objects. Step 3.1: instantiate forwarding ports.

with the outgoing port `cmd` of component `LightCtrl`, is created by sharing `Arbiter`'s `OutPort` instance.

Regarding the FOCUS property that every incoming port is only connected to a single unique sender, two optimizations can be derived:

1. Explicit outgoing port objects are obsolete since connected incoming port objects can be shared with the sender.
2. Forwarding incoming ports of decomposed subcomponents are only needed if two or more receiving incoming ports are connected.

Figure 4.8 depicts the final optimized object structure for the above described example. It can be seen that the incoming port of subcomponent `DoorEval` is also used as incoming port `doorStatus` of the `LightCtrl` component. Subcomponent `DoorEval` uses incoming port

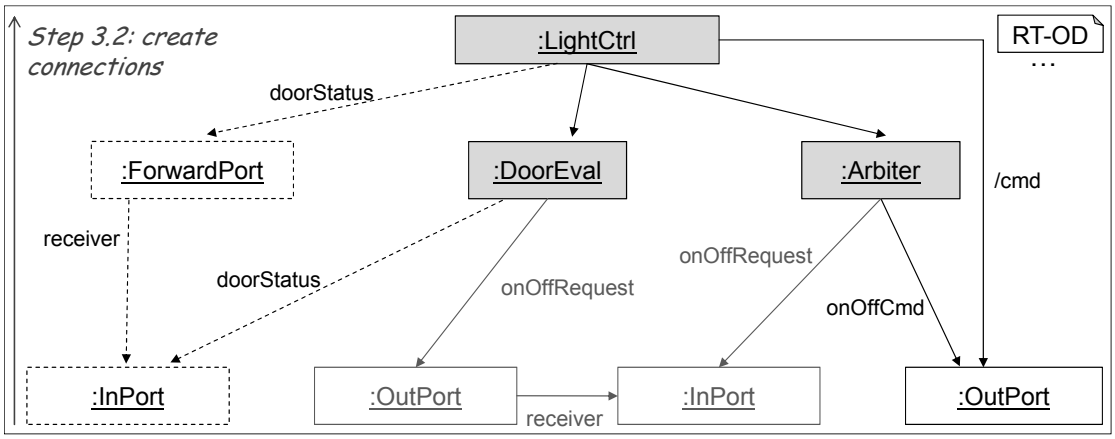


Figure 4.7: Instantiation of components as simulation objects. Step 3.2: create connections.

`onOffRequest`, which belongs to subcomponent `Arbiter`, as its outgoing port. The object that holds the link to `Arbiter`'s port `onOffCmd` respectively `LightCtrl`'s port `cmd` is not depicted in the figure. It is set from the outside when component `LightCtrl` is instantiated in another component in step 3.2. It is not created during the instantiation of the `LightCtrl` object.

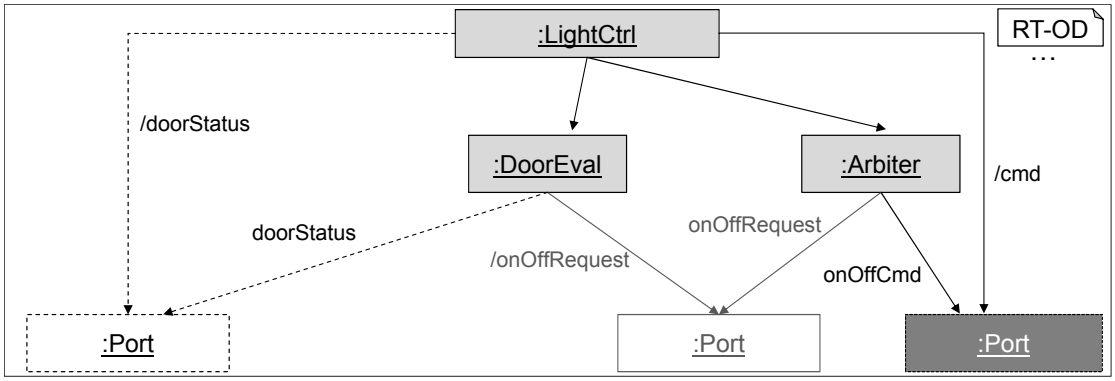


Figure 4.8: Instantiation of components as simulation objects. Optimized object structure.

Regarding these optimizations, the steps that are performed to instantiate components have to be slightly adjusted:

Step 1: Instantiate and link components from top to bottom (unchanged).
Step 2: Instantiate incoming ports of atomic components.
Step 3: For each decomposed component from bottom to top:
  Step 3.1: Create forwarding ports for each incoming port with more than one receiver.
  Step 3.2: Create connections. Therefore, share incoming ports of a receiver as outgoing ports of the connected sender and link forwarding ports to their connected incoming ports.

4.2.2 Simulation Runtime Environment

Based on this optimized object structure, the Java class and interface structure of the RTE is derived. Each MontiArc component definition is represented by an individual Java class that defines the port interface and decomposition of the component. During simulations, it is present as an object which holds the links to port objects that represent the component interface. Links to component objects represent the decomposition of the component. The former are instances of generic runtime classes while the latter are instances of component classes that represent its subcomponents' types. Message flows are simulated in the FOCUS framework. A stream of messages, transmitted over a directed channel which connects incoming and outgoing ports, is suitable to simulate message flows this way. Since such a stream has to transmit both, ticks as well as data messages, a generic message container is needed that is able to represent both. Beside default scheduling strategies and port implementations (see requirement *SRQ9.1*), it is possible to use customized port and scheduler implementations in the simulation (see requirement *SRQ9.2*).

These properties are reflected by the MontiArc RTE. Figure 4.9 depicts the most basic and simplified structure of the RTE. A component only interacts with its environment via its ports. Hence, a component has a set of `outgoing` ports which are used to emit messages to its environment. Further, a component receives messages via its `incoming` ports. It is also depicted that ports are directly connected to their receivers. This is in accordance with the FOCUS property, that a sender can transmit data to a set of receivers. A component can be decomposed to subcomponents. This is reflected by the `subcomponents` aggregation which allows hierarchical component decomposition.

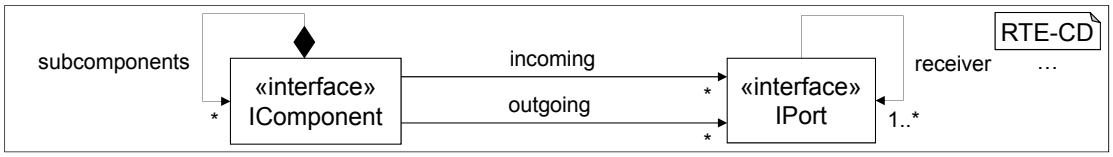


Figure 4.9: Components and Ports in MontiArc's RTE.

Figure 4.10 shows a more concrete MontiArc RTE. Please note that all associations given in Figure 4.9 are also part of the refined RTE diagram but are not repeated. The extended RTE adds *user* (top) and *simulator* (bottom) parts. The former are used by a component developer or a simulation user to implement the behavior of atomic components (see Section 5.5) or to set up a simulation. The latter ones encapsulate simulation specific methods. These are automatically used by the scheduler or the generated component code to schedule messages and simulate time. They are not intended to be used directly by component developers or simulation users. This separation is indicated by a dashed line.

The user relevant parts of the RTE are:

- Interface `IComponent`: represents a component for the component developer or user. It provides a method to set it up.
- Interface `IOutPort`: used by a component developer to emit messages using its `send` method.

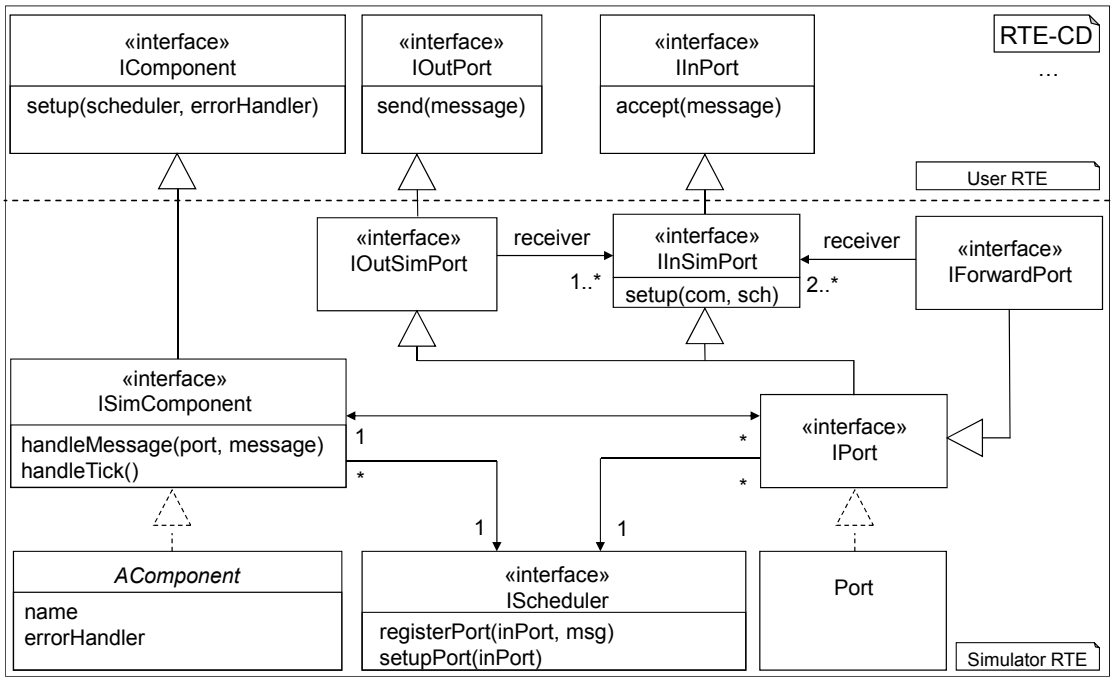


Figure 4.10: MontiArc's RTE. The User RTE parts are used when setting up a simulation or atomic components are implemented. Parts of the Simulator RTE are used by the scheduler and the generated simulation code (see Section 5.4).

- Interface `IInPort`: a component user can trigger a component with messages using the `accept` method of its incoming ports.

The RTE parts used by the Simulator extend the aforementioned parts. They are listed in the following:

- Interface `ISimComponent` encapsulates messages called by the simulation scheduler to trigger component activity. The method `handleMessage(port, message)` is called to process the given data message on port `port`. The method `handleTick()` is called, when a component has to a) increase its internal clock and b) emit a $\surd$ on each outgoing port.
- Abstract class `AComponent` serves as a superclass for generated component classes and contains fields to store the component name and an error handler. The latter can be used in atomic component implementations to, e.g., log misbehavior of components.
- Interface `IOutSimPort` provides methods to add or get receivers and to set or access its associated component.
- Interface `IInSimPort` provides method `setup(com, sch)` to set up an incoming port with the containing component and the corresponding scheduler. This setup method is automatically invoked by the containing component during its own setup. Further scheduling-specific methods to check or influence the state of a port are provided but omitted in the Figure (put to sleep, wake up, and connection state. See Section 4.3).

- Interface `IScheduler` provides a method to set up a concrete scheduler (`setupPort(...)`) as well as a method to trigger scheduling of a certain given port and message (`registerPort(inPort, msg)`). Since the integration of user-defined schedulers is required (cf. requirement *SRQ9.2*), the properties of a simulation scheduler are defined by an interface and not by a concrete class. The concrete default scheduling of the MontiArc simulation is discussed in Section 4.3.
- Interface `IPort` extends `IOutSimPort` and `IInSimPort` and does not add new methods. This unified port interface is needed since connected components share their ports. In this way, the outgoing port object of a sending component can also be used as the incoming port object of the connected receiving component (cf. requirement *SRQ10*).
- Interface `IForwardPort` acts as an incoming port for decomposed components and is forwarding messages to the connected incoming ports of the corresponding subcomponents.
- Class `Port` is the default port implementation used in the simulation. Port objects are instantiated for incoming ports of atomic components. Since the port object of the connected incoming port is used as outgoing port, dedicated objects for outgoing port are not created. For that reason, the port class has to implement the incoming and the outgoing port interface. If more than one incoming port is connected to an outgoing port, the first connected incoming port object is used as outgoing port. Further incoming ports are then added to the `receiver` association.

In MontiArc, it is possible to simulate instant, delaying, synchronous, causal synchronous, and untimed components in parallel (see requirements *LRQ1.1.3* and *SRQ4*). Therefore, the RTE has to distinguish between timed and untimed components. This is needed to prevent untimed components from gaining information about time events. In Figure 4.11 the RTE is extended with Java interfaces and classes which encapsulate methods and attributes of timed components.

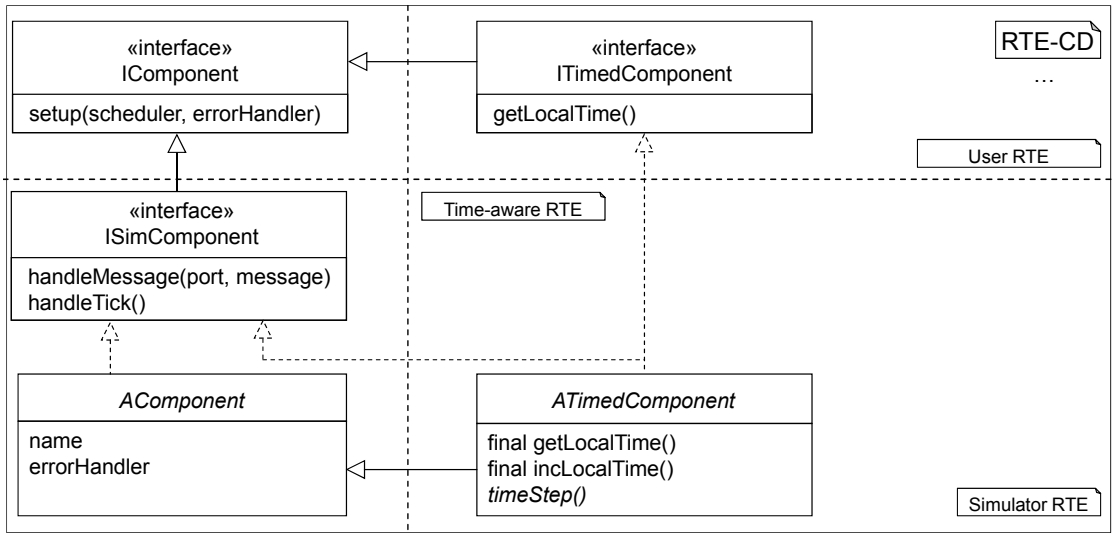


Figure 4.11: Components and Timing in MontiArcs RTE.

These are in the following:

- Java interface `ITimedComponent` provides a method to access the current time of a component.
- Abstract class `ATimedComponent` provides implementations to get and increment the local component time. It also provides the abstract method `timeStep()` which is to be implemented by all timed components. It is called by a timed component itself at the end of the generated `handleTick()` method if the component does not extend another component (see Section 5.4.2). In the case of inheritance, the `handleTick()` method also calls the same method implemented in the superclass. This way, $\surd$ messages are also emitted on the ports defined by the supercomponent.

4.3 Scheduling

To simulate logical distributed and concurrent components in a single thread, an explicit scheduling is needed. The scheduler is responsible for message processing and the simulation of time. Basically, the scheduler decides which component is next to execute and synchronizes incoming data and ticks received on the incoming ports of components to a simulated timed input trace. This trace is propagated to the scheduled components which internally creates timing specific events. These events are processed by the corresponding component implementations. Time domain specific event propagation is discussed in Section 4.4. This section presents, how a scheduler is set up for scheduling and how the default scheduler simulates FOCUS channels with $\surd$ and data messages (cf. requirement *SRQ9.1*).

It is possible to use an individual scheduler for each component (cf. requirement *SRQ9.3*). Therefore the used schedulers have to be properly set up for the components they should schedule. Since scheduling is incoming port driven, a scheduler has to know the components and the corresponding incoming ports it has to schedule (see Figure 4.10). The structure of the default scheduler is depicted in Figure 4.12. It contains the map `comp2Ports` that maps all incoming ports to the containing component. The map `comp2tickfree` contains a set of *tickfree* ports for each scheduled component.

Definition 4.1 ***Tickfree port.*** *Each simulation port at each time has a possibly empty list of incoming events including ticks. This list is used to buffer events which cannot be scheduled immediately. A port is tickfree if its buffer is empty or the first buffered element is a data message.*

Definition 4.2 ***Blocked port.*** *A simulation port is blocked if the first element in its buffer is a* $\surd$.

Since the scheduler processes ticks on all incoming ports of a component simultaneously, ports are blocked when other ports of the same component have not received a $\surd$ yet. If a port is blocked and an event is received, the event is buffered.

Both maps depicted in Figure 4.12 are filled by the method `setupPort(...)` which is called by the scheduled components during the setup of atomic components (cf. Section 5.4.2). Because scheduling depends on the interaction between scheduler and ports, a concrete scheduler is always associated with an `IPortFactory` which produces ports that harmonize with

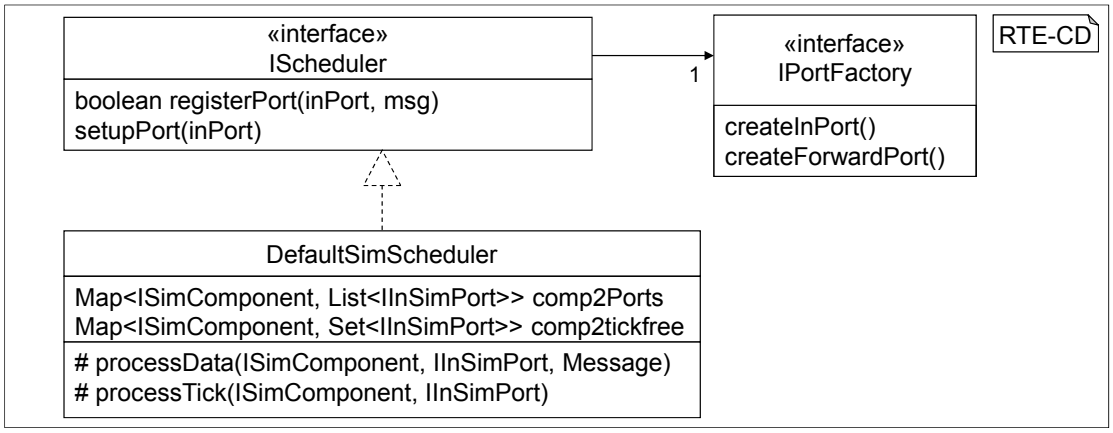


Figure 4.12: Default Simulation Scheduler.

the strategy of the scheduler. The factory is used by the scheduled components during the component setup phase.

During a simulation run, an incoming port that receives a message or tick via its `accept` method, immediately registers itself at its corresponding scheduler using its `registerPort` method if the port is not already scheduled. This can happen if a port is part of a feedback cycle. The boolean result of the `registerPort` method flags whether the given message has been processed immediately or has to be buffered by the port. The following subsections discuss the default scheduling strategy realized by class `DefaultSimScheduler` regarding different scenarios. According to Rumpe [Rum11, Chapter 6], the hexagons used inside the following sequence diagrams contain OCL constraints which are valid below their occurrence. Further, `...` indicates that additional, not modeled communication with other object is possible. A ⓒ tags objects that do not interact with other (not shown) objects after the first and before the last modeled interaction.

The object diagram depicted in Figure 4.13 shows the objects and relations used in the following sequence diagrams. Component `c` has two incoming ports `p1` and `p2` that are scheduled by

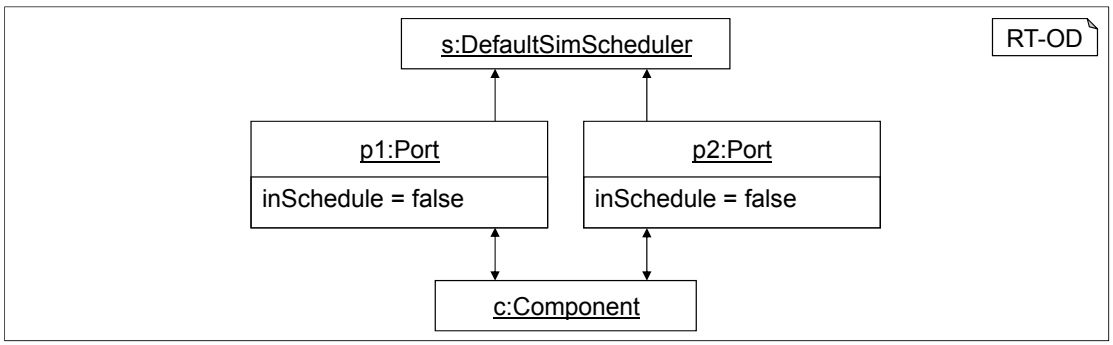


Figure 4.13: Scheduled component `c` and the state of it's ports.

scheduler `s`. Please note that calling the method `handleMessage` of a component propagates the passed data message to the current time interval of the component's input trace. Further, the method `handleTick` finalizes the current time interval of the input trace and causes the component to emit ticks on its outgoing port. There are three main scenarios which are discussed in the following subsections. First, a port that accepts a data message is not involved in an active scheduling process (short: the port is not in a schedule; cf. Section 4.3.1). Second, a port which accepts a $\surd$ message is not in a schedule (cf. Section 4.3.2). And third, a port that accepts either a data message or a $\surd$ is already in a schedule (cf. Section 4.3.3).

4.3.1 Scheduling of Data Messages

Regarding scheduling of data messages, two sub scenarios are possible. The port, on which the message is accepted, is either tickfree or blocked by a $\surd$. Both scenarios are discussed in the following.

Tickfree

Scheduling of data messages on tickfree ports is straightforward as depicted in Figure 4.14. It is performed in the following steps that are illustrated by the sequence diagram:

1. The port (`p1`) accepts a data message (`m`) and checks whether it is currently involved in a scheduling process (`p1.inSchedule`).
2. Since it is not involved (`p1.inSchedule == false`), it registers itself by its scheduler using the `register` method and marks itself as being in a schedule.

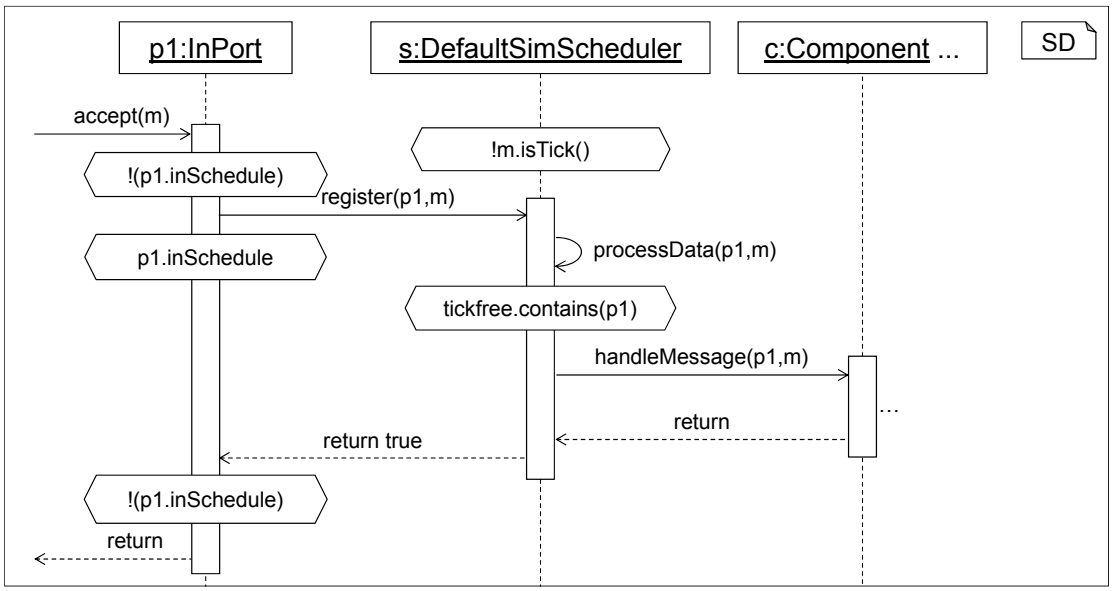


Figure 4.14: Scheduling data messages on a tickfree port.

3. If the registered message is a data message (`!m.isTick()`), the scheduler forwards the call to its `processData` method.
4. It is checked whether the current port is tickfree (`tickfree.contains(p1)`, where `tickfree = comp2tickfree.get(c)`).
5. Since the current port is tickfree, the accepted message is immediately processed by the port's component. Therefore it is passed to the component via the `handleMessage` method.
6. The component then processes the message that usually will result in the emission of messages through its outgoing ports. These messages trigger further scheduling activity which is omitted in the Figure (`...`).
7. When the control flow returns to the scheduler, it returns a `true` to the calling port `p1` to signal that the registered message has been processed. The port now marks itself as not being in a schedule.

Blocked

Scheduling of data messages on a blocked port is depicted in Figure 4.15. Basically, since the port is blocked, the registered message is not processed and has to be buffered by the port for later processing. This is performed in the following steps:

1. The port (`p1`) accepts a data message (`m`) and checks whether it is currently involved in a scheduling process (`p1.inSchedule`).
2. Since it is not involved (`p1.inSchedule == false`), it registers itself by its scheduler using the `register` method and marks itself as being in a schedule.
3. If the registered message is a data message (`!m.isTick()`) the scheduler forwards the call to its `processData` method.
4. It is checked whether the current port is tickfree (`tickfree.contains(p1)`, where

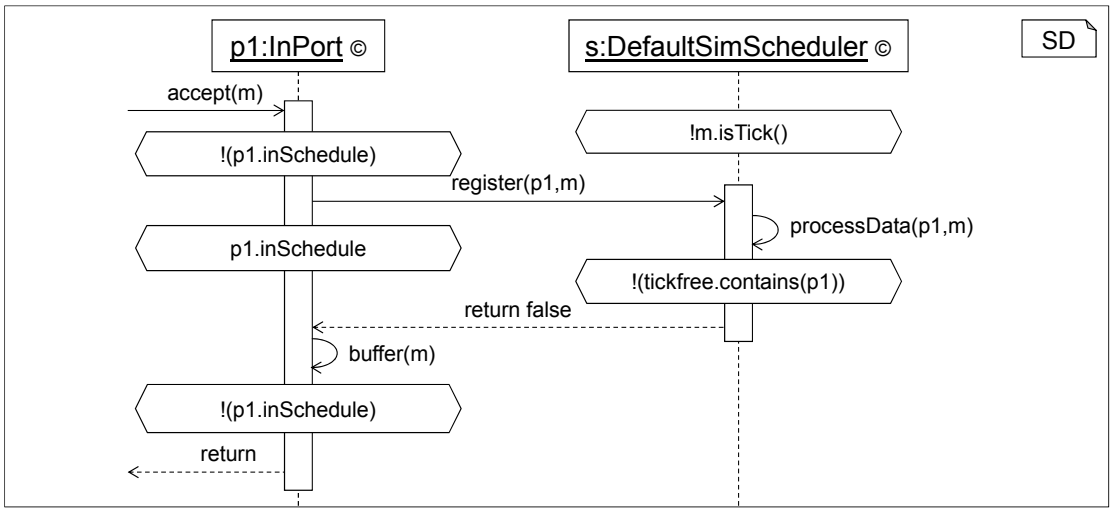


Figure 4.15: Scheduling data messages on a blocked port.

`tickfree = comp2tickfree.get(c)`).

5. Since the current port is not tickfree, the accepted message is not processed yet. The scheduler returns a `false` to the calling port `p1` to signal, that the registered message has not been processed yet.
6. The port buffers the message for later processing (`buffer(m)`) and marks itself as not being in a schedule.

4.3.2 Scheduling of Ticks

When a $\surd$ message is to be scheduled, three sub scenarios are possible. First, the current and some other ports of the component are tickfree (**any tick**). Second, the current port is tickfree but the other ports of the component are not (**final tick**). Third, the current port is blocked but some other ports of the component are tickfree. The last sub scenario is equivalent to the scenario described in Section 4.3.3 and is discussed there. The first two sub scenarios are discussed in the following.

Any Tick

Scheduling of $\surd$ messages on a tickfree port of a component, which has other tickfree ports, is depicted in Figure 4.16. Basically, the current tickfree port is removed from the tickfree set stored in the scheduler. Because the set is not empty after removal, the component has not received a $\surd$ on any of its ports. This is performed in the following steps:

1. The port (`p1`) accepts a $\surd$ (`t`) and checks whether it is currently involved in a scheduling process (`!(p1.inSchedule)`) or whether it is already blocked by a tick (`!(p1.has-`

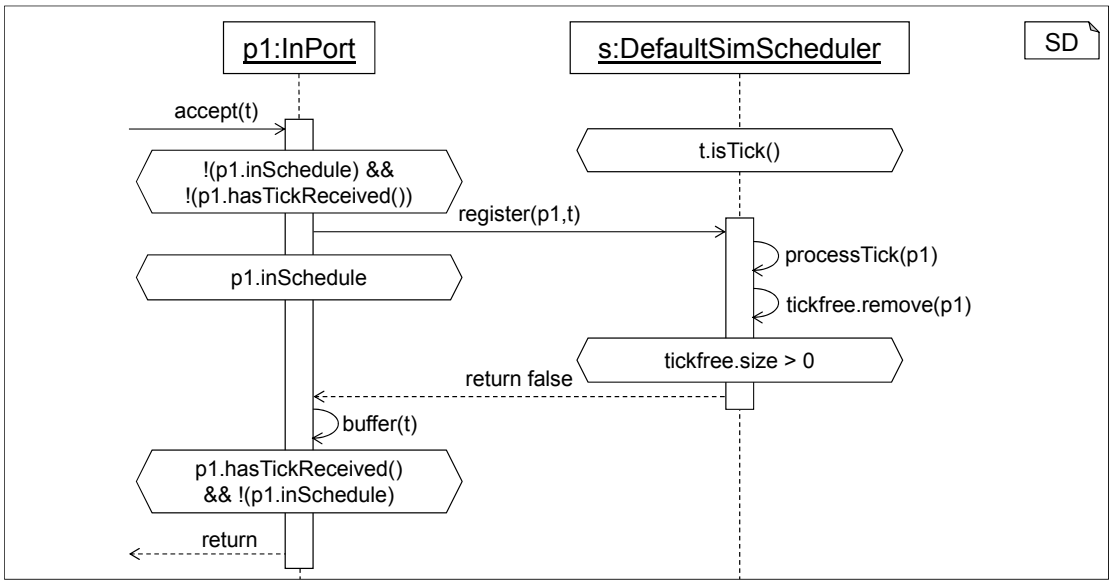


Figure 4.16: Scheduling of any $\surd$ messages.

`TickReceived()`).
2. Since it is not in a schedule or blocked, it registers itself by its scheduler using the `register` method and marks itself as being in a schedule.
3. Since the registered message is a tick (`t.isTick()`), the scheduler forwards the call to its `processTick()` method.
4. The port is then removed from the tickfree set. Since it is not empty afterwards (`tickfree.size > 0`), the scheduler returns `false` to the calling port.
5. The tick is then buffered. This causes the port's method `hasTickReceived()` to return `true` afterwards.

Final Tick of a Time Interval

If the currently scheduled port is the last port contained in the `tickfree` set, its component is allowed to consume a tick on all incoming and emit a tick on all outgoing ports. The scheduling of this process is depicted in Figure 4.17. It is performed in the flowing steps:

1. The port (`p2`) accepts a $\surd$ (`t`) and checks whether it is currently involved in a scheduling process (`!(p2.inSchedule)`) or whether it is already blocked by a tick (`!(p2.hasTickReceived())`).
2. Since it is not in a schedule or blocked, it registers itself by its scheduler using the `register` method and marks itself as being in a schedule.
3. Since the registered message is a tick (`t.isTick()`), the scheduler forwards the call to its `processTick()` method.
4. If the port is the last element in the tickfree set (`tickfree.contains(p2) &&`

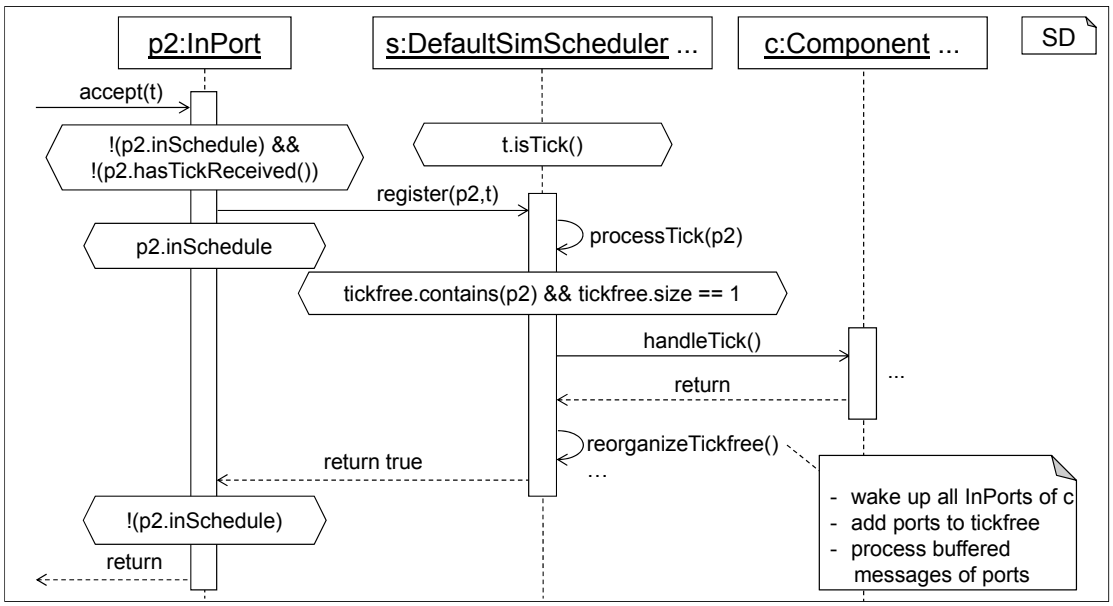


Figure 4.17: Scheduling the final $\surd$ message.

`tickfree.size == 1`), the component `c` has received a tick on each incoming port. The `handleTick()` method is called which emits a tick on each outgoing port and increases the internal clock. Please note that this leads to further message scheduling. The corresponding process is represented by one of the discussed scenarios and is omitted here.

5. When the control flow returns, the tickfree set of `c` has to be reorganized. This is performed in three steps:
   a) Wake up all incoming ports of `c`. This will cause the ports to drop their buffered $\surd$ from the head of their buffers.
   b) Add all incoming ports `p` of `c` back to the associated tickfree set if they have not received a further tick.
   c) Induce all ports in tickfree to process further buffered messages. Please note that this also leads to further message scheduling. This process is also omitted here but exemplary shown in Section 4.3.4.
6. After this reorganization, the scheduler notifies the port that scheduling has been successful by returning the value `true`.
7. The port marks itself as not being scheduled.

4.3.3 Scheduling already Scheduled or Blocked Ports

Scheduling of ports which are blocked by a $\surd$ or that are already in a schedule is straightforward. The performed steps are depicted in Figure 4.18. They are:

1. The port accepts a $\surd$ or data message via its `accept` method.
2. If the port is in an active schedule, immediate message processing is not feasible to avoid recursive endless cycles.
3. If the port is blocked by a tick, it contains a tick at the head position of its buffer. Thus, the message is in the future from the component's point of view. Consequently, it must not be delivered now.
4. In both cases the message is buffered for later processing using method `buffer`.

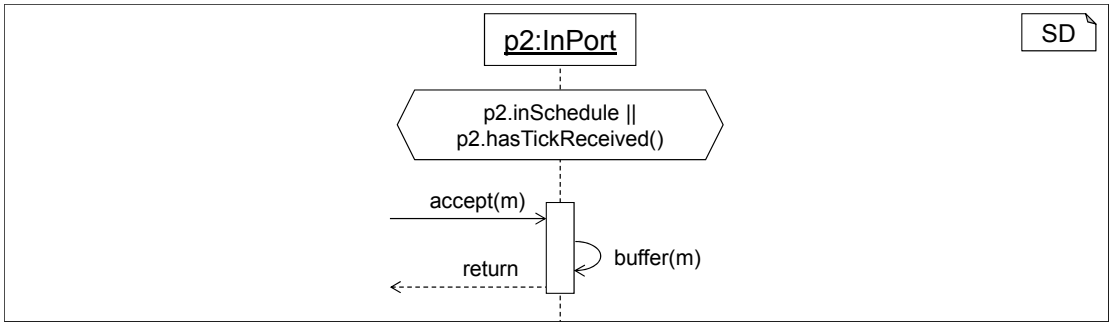


Figure 4.18: Scheduling of data or $\surd$ messages on a port which is already involved in a scheduling process.

4.3.4 Waking up Ports

This scenario illustrates the steps that are performed during the scheduler reorganization. This step is needed to reactivate respectively wake up ports which are blocked by a tick (c. f. Section 4.3.3), after all incoming ports of a component have received a tick. The object diagram in Figure 4.19 depicts the initial object structure. Port `p2` has initially registered at the scheduler with a final tick. It now is `inSchedule` and its message buffer is empty. Port `p1` is currently not in a schedule, is blocked by a $\surd$, and contains further messages {`x`, $\surd$, `z`} in its buffer. Please note that the simulator also allows individual components to be in different time intervals. This is especially helpful when delay should be modeled.

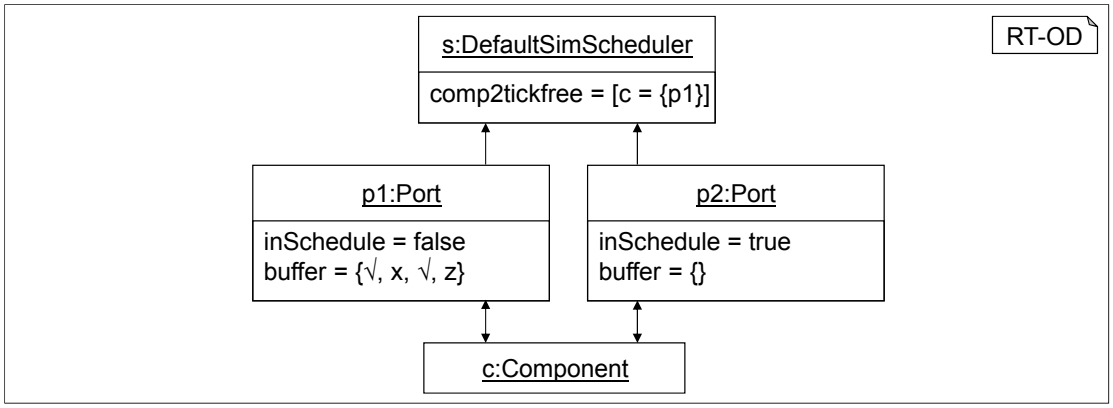


Figure 4.19: Object diagram which depicts the initial situation of the 'waking up ports' scenario.

The sequence of this scenario is given in Figure 4.20. It starts with the call of the `reorganize Tickfree()` method by the scheduler `s` on itself. The following three steps are performed:

1. All ports of the component which belongs to the currently scheduled port (`p2`) wake up. For this purpose, the `wakeUp()` method is called on all incoming ports.
   - Since `p1` has received a tick, it will drop this tick from its buffer.
   - Waking up `p2` does not change its object state since the tick, that it has received, is currently in schedule and therefore not buffered.
2. Then, all ports which have not received a further tick (`!hasTickReceived()`) are added to the components `tickfree` set.
3. Finally, further buffered messages of all tickfree ports are processed by calling their `processBufferedMsgs()` method.
   - This call induces `p1` to register its buffered message `x`. The following message scheduling is performed as described in Section 4.3.1.
   - Since scheduling of `x` has been successful (`return true`), `x` is dropped from the buffer by calling `buffer.poll()`.
   - `p1` then registers the next buffered message. Scheduling of this tick is performed as described in Section 4.3.2. As a result, `p1` has been removed from the tickfree set of component `c` again.

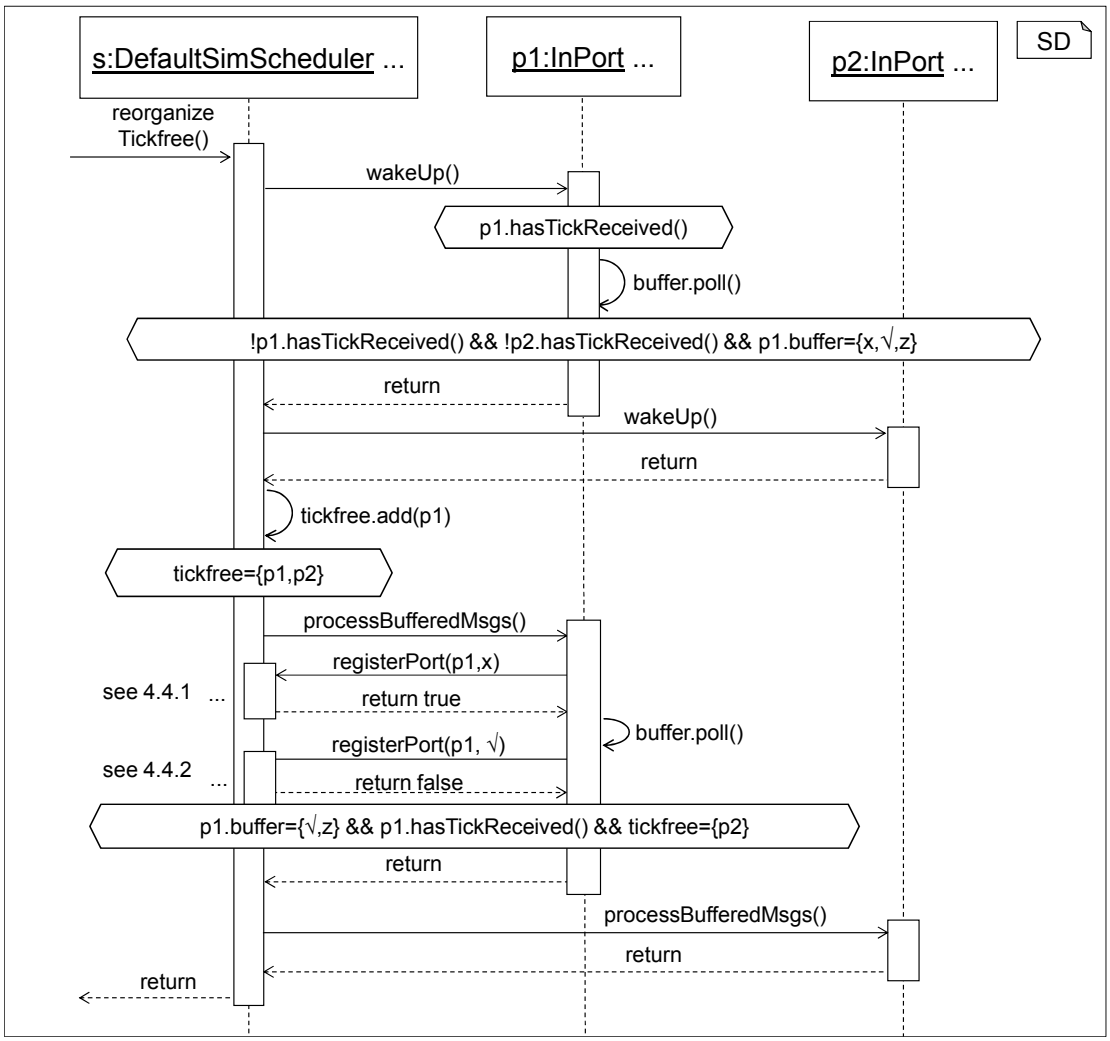


Figure 4.20: Reorganization and waking up of ports.

- Since scheduling of this buffered tick has not been successful (`return false`), `p1` stops to process buffered ticks and the control flow returns to scheduler `s`.
- `s` then tells `p2` to process its buffered messages. Since the buffer of `p2` is empty, the control flow immediately returns to `s` and reorganization is finished.

Waking up ports and processing of buffered messages is performed in a fix order which is determined by the default simulation scheduler. During reorganization, the scheduler iterates ports in the order in which they are added to the scheduler by calling its `setupPort` method. This order corresponds to the order in which the ports are defined in the component model. Hence, the `wakeUp` method is always called on port `p1` before it is called on `p2`. This leads to a deterministic execution of the simulation but also to a preference of ports which are defined first in the model. If this is not adequate for a certain scenario, a custom scheduler can be used.

4.4 Timing Classification

MontiArc and its simulation framework support the simulation of different timing domains that affect the timed behavior of a component (see requirement *SRQ4*). As already mentioned in the Sections 3.3.1 and 3.4.4, components belong to a certain time domain that determines whether or not the component is aware of time and how events are propagated to components. For convenience, a component is qualified with the corresponding time domain in the following. For example, a delayed component belongs to the *delayed* time domain. If no domain is mentioned, an *instant* component is assumed. A component can select one of the five predefined timing domains:

- *Instant* components are time-aware and processes port-specific data events. Their results are emitted without delay.
- *Delayed* components are time-aware and process port-specific data events with processing time (delay ≥ 1 $\surd$). This means that the resulting output is emitted in one or more time intervals later than the input has arrived.
- *Untimed* components are not aware of time and only process port-specific data events.
- Synchronous (*sync*) components are time-aware and synchronously process tuples of data events without delay and without explicit processing of time.
- Causal synchronous (*causalsync*) components are like synchronous components, but have delayed output.

Timed streams are used as the foundation of the MontiArc simulation. Thus, all channels that connect outgoing with incoming ports transmit timed streams. The timing domain of a component determines, how ticks and messages, which are transmitted within these streams, are transformed to events, which are propagated to the component implementation. If a decomposed component combines subcomponents with different timings, the distinct behavior regarding time is automatically unified to the underlying timed stream paradigm. For example, time events are not forwarded to untimed components and only a single message per time interval is forwarded to synchronous components. If a special translation between different timings of subcomponents is required, the model has to be adapted to achieve a smooth interaction. Introducing up- and downscaling subcomponents which translate between different timings in terms of a behavior refinement (see [BS01, Chapter 15]) will then serve as adapters between subcomponents with different timings.

Figure 4.21 outlines the default time-unification process. As discussed in Section 4.3, the `Scheduler` synchronizes the input streams of a component to a timed input trace. For each completed time interval in all input streams, a time event is present in the produced input trace denoted by a $\surd$. All processable data events are also immediately propagated to the component's input trace in order of their occurrence. The scheduler uses the produced timed input trace to trigger the scheduled component. Time events are raised at the component using its `handleTick` method, data events are raised using its `handleMessage` method. The depicted `Specific Event Creation` part of a component then creates timing domain specific events which are passed to the concrete implementation of a component. The predefined timing domains, the corresponding component behavior, and event propagation from timed streams to components are explained in the following.

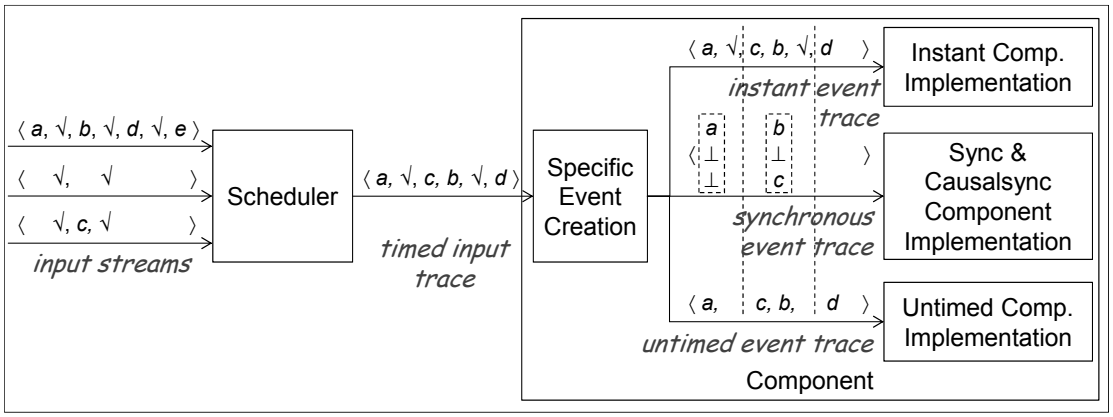


Figure 4.21: Creation of timing domain specific event traces from timed input streams.

4.4.1 Instant Timing

The instant timing domain is used as the *default timing domain* of MontiArc. Hence, if no explicit timing domain is given within a component definition, it is assumed to belong to the instant timing domain. Instant components react to both: the progress of time and separated data messages on each incoming port. As a result of their computation, an instant component can produce arbitrary many output messages during a single time interval on its outgoing ports. The output is produced in the same time interval in which the triggering input occurred. Thus, the behavior of an instant component is *weakly causal*. According to Broy and Stølen [BS01], an input/output behavior is weakly causal if every produced output at each point in time is a result of the so far received input. This way, every output is not affected by future events that have not occurred yet.

The specific event creation of instant components simply forwards the received timed input trace from the scheduler to the component implementation. Thus, the timing domain specific event trace directly corresponds to the timed input trace. Two different kinds of events are processed by an instant component: port-specific data events and time events.

Definition 4.3 ***Data event.*** *A data event is associated with the received payload of a data message and is raised at the incoming port which accepts the data message.*

Definition 4.4 ***Time event.*** *A time event denotes the start of a new time interval of a component. It is raised after the internal clock of a component has been increased and the outgoing ports have emitted a* $\surd$.

Data events are immediately forwarded to the implementation of an instant component. In this way, all data messages within the timed input trace are processed one after another by an instant component. Each raised data event can trigger an instant reaction of the component implementation within the same time interval. If a combination of data events is needed to trigger a specific behavior, the needed event synchronization has to be provided by the component implementation itself.

Input Streams	**Instant Event Trace**
$s_0 = <\ \surd,\ \surd,\ \surd,\ \ldots$ $s_1 = <\ \surd,\ \surd,\ \surd\ >$ $s_2 = <\ \surd,\ \surd\ >$	$t = <\ \surd,\ \surd\ >$
$s_0 = <\ \surd,\ \mathtt{a},\ \mathtt{b},\ \surd,\ \surd\ >$ $s_1 = <\ \surd,\ \mathtt{c},\ \surd,\ \mathtt{d},\ \surd\ >$ $s_2 = <\ \surd,\ \surd,\ \mathtt{e},\ \surd\ >$	$t = <\ \surd,\ \mathtt{a},\ \mathtt{c},\ \mathtt{b},\ \surd,\ \mathtt{e},\ \mathtt{d},\ \surd\ >$
$s_0 = <\ \surd,\ \mathtt{a},\ \surd,\ \mathtt{b},\ \surd,\ \mathtt{c}\ >$ $s_1 = <\ \surd,\ \surd\ >$ $s_2 = <\ \surd,\ \surd\ >$	$t = <\ \surd,\ \mathtt{a},\ \surd,\ \mathtt{b}\ >$

Table 4.22: Exemplary event propagation from timed streams to instant event traces.

Time events denote the start of a new time interval. Thus, they can be used to realize time-based component behavior. Instant components immediately react to triggered time events. Hence, produced output messages are directly emitted in the new time interval.

Some examples for event propagation from timed input streams to an instant component are given in Table 4.22. In the left column, the input streams $s_0 - s_2$ are depicted which are received on the three input ports of the instant component. The right column contains the resulting event traces which are propagated to the implementation of the instant component.

In the first row, propagation of *time events* is demonstrated. Stream s_0 contains at least three ticks, s_1 contains exactly three ticks, and s_2 contains exactly two ticks. Hence, the minimum of completed time intervals is two. This way, two time events are propagated to the component.

The second row demonstrates how *data events* are propagated to an instant component. Two complete time intervals are depicted. In the first time interval, s_0 contains the data messages `a` and `b`, s_1 contains the message `c`, and s_2 is empty. Both streams s_1 and s_2 contain a single message in the second time interval. While solely observing the FOCUS streams of the input ports, no predicates over the order of messages from different streams within the same time interval can be made [BS01]. Thus, the shown event trace is an exemplary valid trace produced from the shown input streams. Nevertheless, the event trace, which is produced by the deterministic default scheduler, preserves the given event order. It contains the corresponding data events `a, c, b` in the first time interval. This implies, that `c` has been received after `a` and before `b`. Also message `e` has been transmitted in stream s_2 before message `d` on stream s_1.

In the third row, it is demonstrated, that only completed time intervals are propagated as time events to the event trace. Stream s_0 contains three completed time intervals, but s_1 and s_2 only contain two completed time interval. Thus, only two time events are propagated to the event trace. Since message `b` is located in the time interval, which started after the last time event, it is also immediately propagated to the input event trace. Message `c` is not propagated since the previous time interval is not closed in s_1 and s_2.

4.4.2 Delayed Timing

The event propagation of delayed components is the same as for instant components. As well as instant components, they can react to time and data events. In contrast, delayed components are *strongly causal*. According to Broy and Stølen [BS01] this especially means, that produced output of a strongly causal component only depends on input received in past time intervals. For example, an output produced in time interval t only depends on input received until time interval $t-1$. Since this restriction only affects the output behavior of a component, event propagation from input streams to the component event trace follows the same rules that are described in the previous section. Delayed components are suitable to model processing time of a component and to avoid the Brock-Ackerman anomaly [BA81] in feedback cycles (see Section 3.5.5). In MontiArc, delayed components can be easily constructed by combining an instant component with predefined `Delay` components from the MontiArc library (see Section 6.8.2) in a decomposed component.

4.4.3 Untimed

Untimed components are not aware of timing events but can react to data events. The implementations of these components are still weakly causal since they cannot produce output based on future input. Data events are propagated to untimed components just like to instant or delayed components. In contrast to the latter, time events are not propagated to untimed components. Consequently, they cannot react to the progress of time. This means, that the ticks within the timed input event trace, which is produced by the scheduler, are filtered out to produce an untimed event trace.

Exemplary event propagation from timed input streams to untimed components is demonstrated in Table 4.23. In row one it can be seen, that no time events at all are propagated to an untimed component. Please note, an untimed component still synchronously consumes ticks and emits ticks on its outgoing ports (see Section 4.3.2). However, the component implementation is

Input Streams	**Untimed Event Trace**
$s_0 = <$ √, √, √, ... $s_1 = <$ √, √, √ > $s_2 = <$ √, √ >	$t = <>$
$s_0 = <$ √, a, b, √, √ > $s_1 = <$ √, c, √, d, √ > $s_2 = <$ √, √, e, √ >	$t = <$ a, c, b, e, d >
$s_0 = <$ √, a, √, b, √, c > $s_1 = <$ √, √ > $s_2 = <$ √, √ >	$t = <$ a, b >

Table 4.23: Event propagation from timed streams to untimed event traces.

not aware of this procedure and cannot react to time events. In this way, the untimed component in our example will emit two ticks on its outgoing ports for the given input streams in row one. Row two demonstrates that data events are propagated in order of their arrival and according to the containing time intervals. The last row of the table demonstrates that only data events from completed or started time intervals are propagated to an untimed event trace. Basically, untimed event traces correspond to instant or delayed event traces with filtered out ticks.

4.4.4 Synchronous Timing

Synchronous components are timed components which synchronously process data events with a weakly causal implementation. Synchronous components are inspired by time-synchronous FOCUS streams and therefore are able to process at max one input event at each time interval. Further, synchronous components only sent at max one message per outgoing port as a reaction to the received input event. The propagation of messages from timed input streams to synchronous event traces defines the semantics of synchronous components which operate on timed streams.

Definition 4.5 ***Data event tuples.*** *A data event tuple (DET) is an ordered n-tuple of data events which is propagated from input streams to synchronous event traces at the end of each time interval. The tuple is constructed by taking a single data message from the current time interval of each input stream. If a stream contains multiple messages within the current time interval, the last message is used to construct the tuple.* $\perp$ *denotes, that no message is transmitted within the corresponding time interval. A DET further denotes the end of a time interval and thus is also interpreted as a time event.*

To simplify the implementation of synchronous components, messages of all input streams are condensed to a single *data event tuple* for every time interval. In this way, synchronous implementations do not have to synchronize incoming messages themself.

In Table 4.24 some examples for event propagation from timed input streams to synchronous event traces are given. In the first row, propagation of *time events* is demonstrated. The minimum of time intervals in the input streams is two. Hence, two time events are propagated to the component. In contrast to instant or delayed components, no explicit time events are propagated to a synchronous event stream. Rather, a DET is constructed which also denotes the end of a time interval. Since no messages are contained in the input streams, the constructed DETs do not contain messages, which is denoted by a $\perp$. Please note, the first depicted DET corresponds to the end of the zeroth time interval.

The second row demonstrates regular DET construction. Time interval one of input stream s_0 contains message `a`, stream s_1 contains a `b`. The corresponding DET contains message `a`, `b`, and $\perp$ since stream s_2 does not contain a message in this time interval.

The third row demonstrates another difference between data events and DETs. In the previous examples (see Table 4.22 and 4.23), the data event, which represents message `b` from the second time interval of stream s_0, is immediately propagated to the respective event trace. In contrast, DETs are created when a time interval is completed on all input streams. Thus, a DET, which contains message `b`, is not (yet) created in the synchronous event stream.

Input Streams	**Synchronous Event Trace**
$s_0 = <$ √, √, √, ... $s_1 = <$ √, √, √ $>$ $s_2 = <$ √, √ $>$	$t = < \begin{pmatrix}\bot\\\bot\\\bot\end{pmatrix}, \begin{pmatrix}\bot\\\bot\\\bot\end{pmatrix} >$
$s_0 = <$ √, a, √, √ $>$ $s_1 = <$ √, b, √, c, √ $>$ $s_2 = <$ √, √, d, √ $>$	$t = < \begin{pmatrix}\bot\\\bot\\\bot\end{pmatrix}, \begin{pmatrix}a\\b\\\bot\end{pmatrix}, \begin{pmatrix}\bot\\c\\d\end{pmatrix} >$
$s_0 = <$ √, a, √, b, √, c $>$ $s_1 = <$ √, √ $>$ $s_2 = <$ √, √ $>$	$t = < \begin{pmatrix}\bot\\\bot\\\bot\end{pmatrix}, \begin{pmatrix}a\\\bot\\\bot\end{pmatrix} >$

Table 4.24: Exemplary event propagation from timed streams to synchronous event traces.

Please note that the MontiArc simulation uses timed streams. Thus, it is possible that a port of a time-synchronous component can be stimulated with more than a single data message in a time interval. Since this is actually illegal in the time-synchronous FOCUS domain, a warning is emitted by the simulator and the last message of the time interval is used to construct the DET. The last message is used because it is the most recent message which has been produced by the potentially faster environment of the synchronous component. If for example s_0 in the second row contains the messages `a` and `z` in the first time interval, `z` is used for DET construction.

4.4.5 Causal Synchronous Timing

Causal synchronous components are synchronous components with a *strongly causal* implementation. They also react to DETs which uniformly model the progress of time and acceptance of data. Just like delayed components (see Section 4.4.2), their produced output solely depends on input received in past time intervals. Again, this restriction only affects the output behavior of a causal synchronous component. Thus, event propagation from input streams to the causal synchronous event trace is equivalent to the propagation presented in the previous section. Just like delayed components, causal synchronous components are suitable to model processing time of a component and to avoid the Brock-Ackerman anomaly [BA81] in feedback cycles (see Section 3.5.5).

4.4.6 Timing Domain Overview

Table 4.25 summarizes the presented timing domains and grants an overview of their properties.

4.5 Optimization and Runtime Measurement

Motivated by requirement *SRQ10*, the scheduling strategy presented in Section 4.3 is compared with a simple round robin scheduler (RRS) regarding the simulation runtime. Since the Mon-

Timing Domain	Time Aware	Event Processing	Causality
Instant	×	data events	weak
Delayed	×	data events	strong
Untimed		data events	weak
Synchronous	×	data event tuple	weak
Causal synchronous	×	data event tuple	strong

Table 4.25: Properties of MontiArc's timing domains.

tiArc scheduler approves to be substantially faster, further optimization potential is identified and scheduler variants are derived. These variants are then compared regarding their runtime performance in an identical scheduling scenario.

4.5.1 Simple Round Robin Scheduling

A simple round robin scheduler (RRS) schedules the simulation according to the following steps. If a message is to be sent, the transmitting port registers itself at the scheduler and the control flow returns to the sending component. The RRS is periodically triggered by a simulator that causes the RRS to process the next registered port. In this way, the scheduler iterates over all registered incoming ports and triggers the corresponding component to handle data or time processing depending on the current message. The component then has to decide itself, whether it is able to propagate the current event to its behavior implementation. For that purpose, the buffers of the incoming ports have to be checked for blocking ticks. If the component produces further events while processing a message, the corresponding receiving ports are registered at the scheduler. Since the receiving component is not activated by the scheduler, these events are not processed although the receiver might be able to.

This solution has several drawbacks that measurably slow down the simulation:

1. Synchronization of ticks is handled by the component itself by simply checking on every received $\surd$ whether all other ports have received a $\surd$, too. This leads to repeated useless calls of the component with expensive iterations over all incoming ports and the corresponding buffer.
2. The scheduler chooses the next port to process in a round robin way by simply selecting the next port of all registered ports. Even if this port is been blocked by a $\surd$, the scheduler passes it to the component that itself has to decide whether the message on this port can be processed or not.
3. Separation of scheduler and simulator leads to further overhead needed to recognize, whether the current simulation is finished.

These drawbacks are avoided by the MontiArc scheduler presented in Section 4.3 with the following optimizations:

1. Synchronization is performed by the scheduler and not the component. By remembering which ports are blocked and which are tickfree, useless calls of the component are avoided.

Also separated methods for $\surd$ and data handling in the component are introduced, so the scheduler can directly call the appropriate method for the currently scheduled message.
2. Components are solely triggered with tickfree ports which are ready to process a messages (see Definition 4.1). Hence, useless component calls are avoided.
3. Time handling of components is solely triggered by the scheduler if all corresponding incoming ports have received a tick. Thus, useless component calls are avoided.
4. By greedily scheduling newly registered ports, if possible immediately, an extra simulator is not needed. Blocked ports, which cannot be scheduled now, are reactivated after processing the blocking tick. This way, the control flow overhead of the simulator is omitted.

The RRS and the MontiArc scheduler are compared using the components depicted in Figure 4.26. The `Inner` components replicate each received message on port `inputX` `X` times and emit the replicas on port `outputX`. This way, a data message received on port `input2` is emitted twice on port `output2`. Thus, an asynchronous workload of messages which are to be scheduled is created on the incoming ports of the subcomponents.

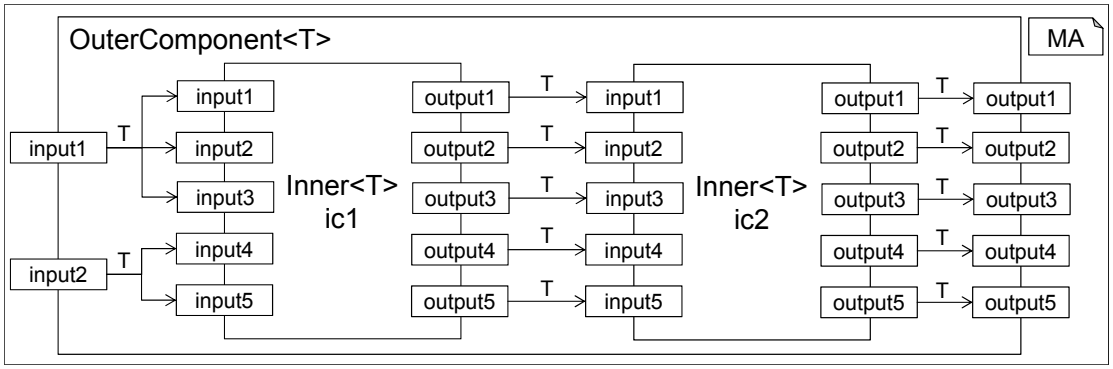


Figure 4.26: Component setup used to compare the RRS and the presented MontiArc scheduler.

The shown setup is executed 50 times with three different setups while recording the execution time. Afterwards the average execution time of all 50 runs is compared. The measurements are executed on an Intel® Core™2 Duo CPU T9800 @ 2,93 GHz, 64 Bit. The setups are:

1. `Only Ticks`: Scheduling of up to 144.000 ticks as input on both incoming ports of the `OuterComponent`.
2. `Sync`: Alternate scheduling of 97.900 ticks and the same amount of data messages, where each data and $\surd$ message has been send simultaneously on port `input1` and `input2` of the `OuterComponent`.
3. `Async`: Asynchronous sending of ticks with up to 109.224 messages. On port `input1`, each 4^{th} message is a $\surd$. On port `input2`, each 9^{th} message is a tick. Since the RRS only stops the simulation, if all messages are processed, an even amount of ticks has to be send on both incoming port. Thus, the setup stops to send ticks on port `input1` after 12.136 ticks ($109.224/9$) and only sends ticks on port `input2`.

The results of this comparison are depicted in the diagram in Figure 4.27. The horizontal axis is labeled with the total number of scheduled data and tick messages (`Number of Mes-`

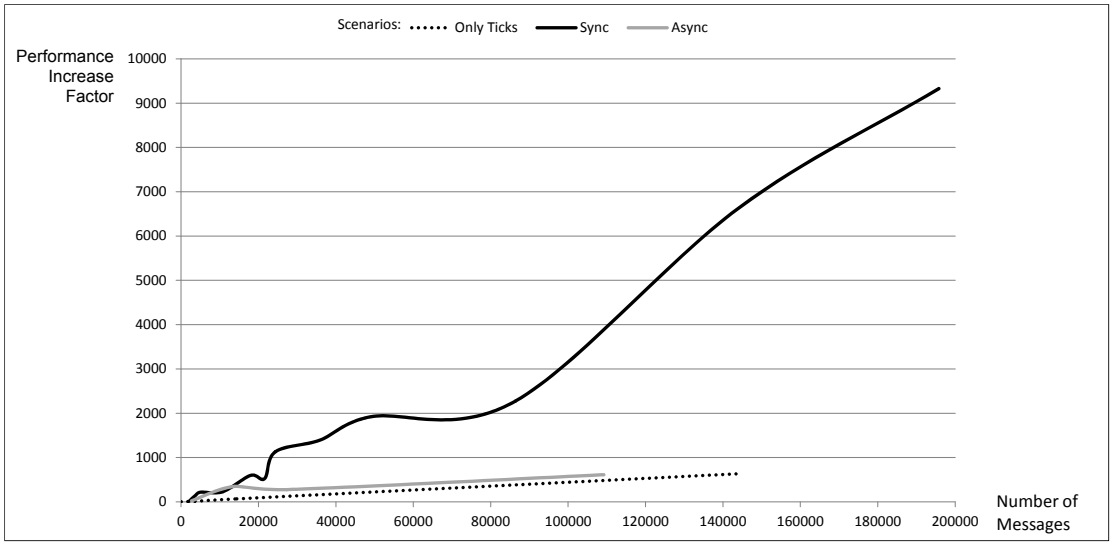


Figure 4.27: Comparison results: Round robin vs. MontiArc scheduler. The `Performance Increase Factor` is the quotient of the average execution time of the RRS and the average execution time of the MontiArc scheduler. `Number of Messages` = Number of ticks + Number of data messages.

sages) that are send to the `OuterComponent`'s incoming ports. The vertical axis is labeled with the `Performance Increase Factor` which is the quotient of the average execution time of the RRS and the average execution time of the MontiArc scheduler. It denotes how much faster the latter is in comparison to the RRS.

The `Only Ticks` scenario, with a maximal amount of 144.000 ticks, is finished by the RRS in an average time of 77.790 ms. The MontiArc scheduler only needs 123 ms and thus is about 634 times faster. The `Async` scenario with 109.224 messages is executed in an average time of 72.279 ms by the RRS and 118 ms by the MontiArc scheduler. Thus, the latter is about 613 times faster. The most noticeable difference is measured in the `Sync` scenario with a total amount of 195.800 messages. This scenario is scheduled in an average time of 316.258 ms by the RRS, the MontiArc scheduler only needs about 34 ms. Thus the latter is up to 9.329 times faster.

This massive speed up, especially of the last scenario, is explained by having a detailed look at the setup. The RRS only processes a single port once and then processes the next registered port in his round robin schedule. The used components lead to an asynchronous workload between ports `input1` to `input5` of the inner components. If one data message is send via port `input1` and `input2` of the outer component, a single message is emitted on its outgoing port `output1` but 25 messages on port `output5`. Thus, the RRS has to iterate up to 25 times through its round robin schedule to process all messages on this port. This renders the scheduler to be very inefficient and slow.

Since all ports in this scenario are never blocked by a tick, data messages are immediately processed by the MontiArc scheduler and greedily scheduled as far as possible. Thus, data messages are immediately pushed through the complete path from the `OuterComponent`'s

incoming to its outgoing ports. Additionally no synchronization overhead for $\surd$ messages is rendered since ticks are always sent synchronously to both input ports.

4.5.2 Further Optimization potential

Even though the MontiArc scheduler is substantially faster than the round robin scheduler (RRS), there is still some potential for further optimizations. While sticking to the described scheduling strategy (see Section 4.3), further optimizations are identified by analyzing the internal data structures within the implementation of the scheduler. The resulting nine scheduler variants are listed and explained in Section 4.5.3. As depicted in Figure 4.12 on page 97, the scheduler uses two maps (`HashMap`) to map a list (`LinkedList`) of incoming ports and the set (`HashSet`) of tickfree ports to the corresponding component. The following operations, the names are printed in bold font for later referencing, access these data structures:

- Data messages scheduling (method `processData()`):
  - **resolveTickfree**: Set of tickfree ports is resolved from the second map using its `get(IComponent)` method. This method then computes the hash value of the passed component to identify the related value. The rest of the get method execution can be neglected (O(1)).
  - **isPortTickfree**: The resolved set of tickfree ports is accessed by using its `contains(IPort)` method. Since a `HashSet` is used that internally stores its values in a `HashMap`, it also calculates the hash values of the passed ports. The rest of the contains method execution can be neglected (O(1)).
- $\surd$ message scheduling (method `processTick()`):
  - **resolveTickfree** (see above).
  - **markPortBlocked**: The tickfree set is accessed once with its `remove(IPort)` method (hash value calculation + O(1)).
  - **areAllPortsBlocked**: The tickfree set is further accessed once with its `isEmpty()` method (O(1)) to check, whether all ports of the component received a $\surd$ yet.
  - **reorganize**: Afterwards, the list of all ports is iterated to reorganize the tickfree data structure of the scheduler (see Section 4.3.4). `LinkedList` iteration is performed in O(n) and thus depends on the amount of scheduled ports. The tickfree data structure has to be adjusted for each port in this iteration. E.g., ports that have not received a further tick are added to the tickfree `HashSet` again.

Summing up, using these HashMap based data structures, that actually have a constant complexity of O(1) for their get, contains, and remove methods, additionally produce some overhead for hash value calculation that can be reduced. Nevertheless, the `isEmpty()` method, that is used to check whether all ports of a component already received a tick, is realized very performant.

4.5.3 Scheduler Variants

In the following, several variants of the presented scheduler are discussed that replace the mentioned `HashMap` respectively `HashSet` based data structures with alternative data structures to

avoid the drawback of hash value calculation. To distinguish the presented MontiArc scheduler from the newly introduced variants, it is named **HashSet** scheduler in the following.

BoolArray

The `HashSet` used to store tickfree ports is replaced with a Boolean array that flags for each scheduled port whether it is tickfree. Therefore, the scheduler has to assign a component wide unique number to the ports during the component setup. This way, it later is able to identify the related position in the Boolean array. Thus, operation **markPortBlocked** is performed in a single operation by switching the position in the array from `true` to `false`.

BitSet

The `HashSet` is replaced with a `BitSet` that also uses the port id (see above) to identify whether a port is tickfree. The `BitSet` implementation provided by Java realizes a vector of bits. Each bit can be examined, set, or cleared individually.

PortMap

The `HashSet` is replaced with a `PortMap`, a custom data structure which combines a Boolean array with a counter that monitors the amount of tickfree ports.

Replace HashMaps

The `HashMaps` used to, e.g., map a component to its tickfree ports, are replaced with `ArrayLists`. Thus, similar to ports, the scheduler assigns a unique identifier (non negative integer) to each scheduled component during the setup. The data structure to handle tickfree ports of a component and the list of all component's ports are then stored at the position in the ArrayList that corresponds to the id of the component. This way, hash value calculation is omitted. Several subvariants of this variant exist that also use the previously described optimization alternatives. Namely these are:

- HashSetNoMaps,
- BoolArrayNoMaps,
- BitSetNoMaps, and
- PortMapNoMaps.

Generated Component Tailored Scheduler

A specialized scheduler generated for a component type. Each instance of a component type uses its own scheduler instance. This way, a mapping from component instance to the corresponding tickfree ports is totally omitted. The ports of the component respectively links to the ports are directly stored within the scheduler to avoid further general data structures. The information, whether a port is tickfree, is stored within a Boolean array. The scheduler generator has to associate each port with a fix position in this array.

Two variants of this scheduler are compared. One, that is directly generated into the component implementation (**Internal**), and an external one, that is created and passed to the component by the corresponding component factory (**External**). Please note, these schedulers are handwritten to demonstrate the capabilities of generated schedulers. Actually, a scheduler generator has not been realized in this thesis.

4.5.4 Comparison Setup

The discussed scheduler variants are compared by scheduling two different component setups with various scheduling scenarios. Please note that this comparison uses more complex and thus more computation intensive scenarios and setups than the comparison described in Section 4.5.1. The change of the setup is motivated by the fact that the compared schedulers are much faster than the RRS. This way, more significant differences between the distinct schedulers are to be expected.

The first component setup is given by component `LoadTest_2_50` which is depicted in Figure 4.28. It contains a linearly connected chain of 50 `LoadTestInner2` subcomponents. Each `LoadTestInner2` component simply forwards a message accepted on an incoming port to an outgoing port. This way, the functionality of the component only needs minimal computation time. Thus, a message sent to port `sIn1` of component `LoadTest_2_50` is emitted on port `sOut1` after being passed through all subcomponents. This component setup focuses on a higher amount of subcomponents with a lower amount of ports (100 in sum).

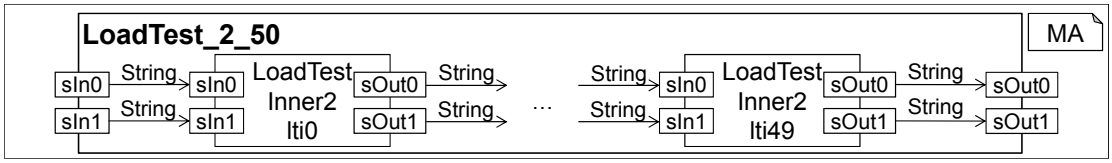


Figure 4.28: Component `LoadTest_2_50` used to compare the discussed scheduler variants in a setup with many subcomponents with few ports.

The second setup is given by component `LoadTest_100_8` which is depicted in Figure 4.29. It contains a linearly connected chain of eight `LoadTestInner100` subcomponents. Again, this component type simply forwards messages received on port `sInX` to port `sOutX` ($X \in \{0 \dots 99\}$) to minimize needed computation time within the components. This setup focuses on a higher amount of ports (800 in sum) with a lower amount of subcomponents. Since this setup is more complex than the first one, it is only executed for a selected set of scheduler candidates from the first executed setup.

The distinct scheduler scenarios used to compare the scheduler variants differ in the frequency of sent ticks (*Tick Frequency, TF)*). Starting with a frequency of 64, every 64^{th} message is a tick, the TF is halved until only ticks are sent. In sum, 100.000 messages (data or tick) are sent on every incoming port which results in 10.000.000 scheduling requests in the first and 80.000.000 scheduling requests in the second setup. Each scenario is repeated 100 times while measuring the CPU time that is needed to schedule a complete scenario in milliseconds using method `ThreadMXBean.getCurrentThreadCpuTime()`. Afterwards,

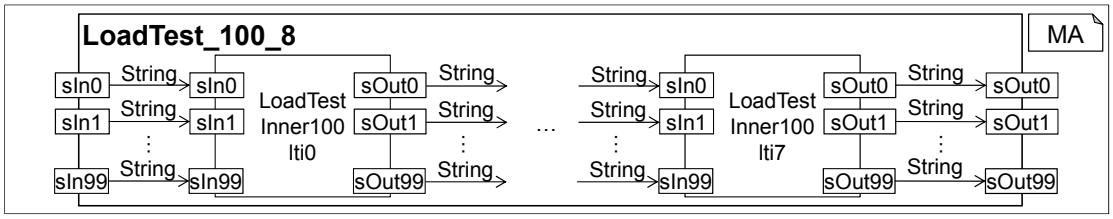


Figure 4.29: Component `LoadTest_100_8` used to compare the discussed scheduler variants in a setup with few subcomponents that have many ports.

the average execution time of all 100 executions is stored. The measurements are again executed on an Intel® Core™2 Duo CPU T9800 @ 2,93 GHz, 64 Bit. Please note that method `getCurrentThreadCpuTime()` has a measured accuracy of about 15.6 milliseconds on the mentioned system. Nevertheless, using this method is more accurate than using a wall clock (`Systen.currentTimeMillis()`) which is also influenced by other processes running on the system. However, the measured values can vary 15.6 milliseconds in the one or the other direction. This effect is counteracted by the repeated execution of each setup. Please also note that assertion checks have been disabled to avoid additional overhead in BitSet based schedulers.

4.5.5 Results

The measured results of the first setup are depicted in Figure 4.30. The contained tables group schedulers in blocks which contain a) schedulers that use HashMaps to map components to their tickfree ports (HashSet, BoolArray, BitSet, and PortMap), b) schedulers that use ArrayLists to map components to their tickfree ports (HashSetNoMaps, BoolArrayNoMaps, BitSetNoMaps, and PortMapNoMaps), and c) generated schedulers (Internal and External). In each block the best values are highlighted in light gray, the worst are highlighted in dark gray. The upper table contains the average execution times for the different tick frequencies, the lower table contains the ratio between the best scheduler in the current category and the current scheduler in percent. For example scheduler HashSet is 29,95 % worse than scheduler PortMap for a tick frequency of 64. The average execution time of the fastest and slowest schedulers of each group is also depicted in the diagram on the right side of the figure.

It can be seen that for almost all tick frequencies the PortMap scheduler is the best in its category. Only for a TF of one, which means only ticks are processed, it is minimal worse (37 ms; 3,97 %) than the BoolArray scheduler. As expected, the HashSet scheduler is indeed the slowest for almost every TF. Only for a TF of 64, the BitSet scheduler is slightly slower (3 ms). This, however, lies below the accuracy of the performed measurement and thus can be neglected.

The PortMapNoMap scheduler is constantly the fastest in its category, while the HashSetNoMaps is the worst. It is between 29,90 % and 62,34 % slower than the PortMapNoMap scheduler. Nevertheless, every scheduler that uses an ArrayList to map components to their tickfree ports (group b) is faster than the related scheduler that uses a HashMap for this purpose (group a).

Both generated schedulers are faster than the generic ones from group a) and b), while the

	a)				b)				c)	
TF / ms	HashSet	BoolArray	BitSet	PortMap	HashSetNoMaps	BoolArrayNoMaps	BitSetNoMaps	PortMapNoMaps	Internal	External
64	684	629	687	479	571	561	400	400	304	275
32	703	629	691	486	590	564	420	403	309	280
16	746	640	703	504	632	571	459	415	321	291
8	831	658	722	537	716	585	540	435	341	318
4	998	699	757	602	885	612	700	477	385	367
2	1340	773	834	728	1223	662	1013	559	473	461
1	2008	906	969	943	1887	739	1608	711	610	646

	a)				b)				c)	
TF / Ratio	HashSet	BoolArray	BitSet	PortMap	HashSetNoMaps	BoolArrayNoMaps	BitSetNoMaps	PortMapNoMaps	Internal	External
64	-29,95	-23,86	-30,23	0,00	-29,90	-28,72	-0,01	0,00	-9,79	0,00
32	-30,90	-22,73	-29,66	0,00	-31,62	-28,45	-4,03	0,00	-9,43	0,00
16	-32,50	-21,25	-28,34	0,00	-34,41	-27,40	-9,72	0,00	-9,40	0,00
8	-35,37	-18,39	-25,63	0,00	-39,15	-25,56	-19,29	0,00	-6,57	0,00
4	-39,70	-13,98	-20,52	0,00	-46,07	-21,93	-31,75	0,00	-4,54	0,00
2	-45,67	-5,87	-12,66	0,00	-54,25	-15,45	-44,74	0,00	-2,49	0,00
1	-54,90	0,00	-6,54	-3,97	-62,34	-3,89	-55,82	0,00	0,00	-5,63

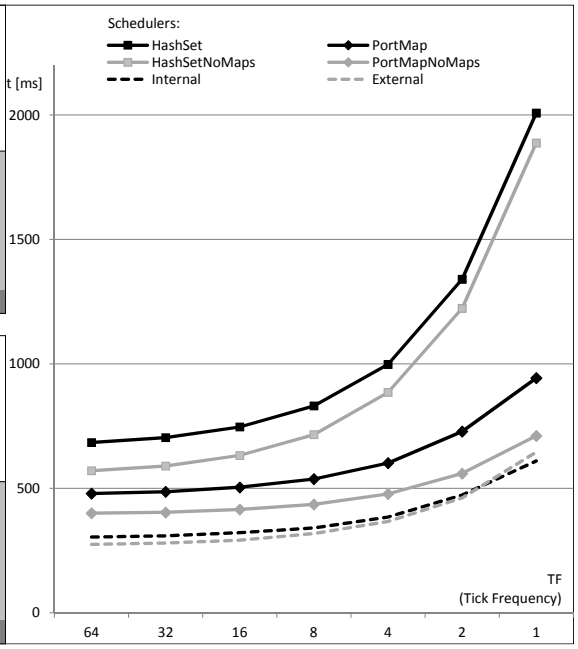


Figure 4.30: Scheduler comparison results of the average execution time in milliseconds for component LoadTest_2_50. The tick frequency of the distinct scenarios is ranging from 64 to one.

external scheduler is slightly faster for tick frequencies from 64 to 2 (between 9,79 % and 2,49 %) and slightly slower for a TF of 1 (5,63 %).

For the second setup, the following schedulers are selected. All schedulers from group b) except the worst one (HashSetNoMaps), which has been even slower than the best scheduler from group a). As a representative of group a) the BoolArray scheduler is used with the fastest results for a TF of 1. Generated schedulers are not selected since they have already proven to be substantially faster than generic schedulers.

The results of the second setup that evaluates the schedulers BoolArray, BoolArrayNoMaps, BitSetNoMaps, and PortMapNoMaps are depicted in Figure 4.31. Again, the table in the top-left depicts the average execution times in milliseconds, the bottom-left corner contains a table with the calculated ratio between the current and the best scheduler, and the diagram on the right-hand side visualizes the average execution times of the evaluated schedulers.

It can be seen that the BoolArray scheduler is the slowest for all TFs, its ArrayLists based variant BoolArrayNoMaps can be regarded as the second slowest scheduler in this setup. For TFs of 64, 32, and 16 the BitSetNoMaps scheduler is the fastest. Nevertheless, for these TFs, all ArrayList based scheduler are pretty close together since the difference in execution time is below 2 percent. For the TFs of 8, 4, 2, and 1, the PortMapNoMaps scheduler is the fastest one which stands out from the other schedulers. For the most tick-intensive scenario it has an improvement rate ranging from 15,89 % to up to 39,15 %.

TF / ms	BoolArray	BoolArrayNoMaps	BitSetNoMaps	PortMapNoMaps
64	5977	5857	5855	5931
32	6057	5921	5879	5960
16	6248	6041	5973	5994
8	6623	6270	6148	6082
4	7344	6742	6475	6239
2	8744	7671	7127	6540
1	11323	9231	8192	6890

TF / Ratio	BoolArray	BoolArrayNoMaps	BitSetNoMaps	PortMapNoMaps
64	-2,03	-0,03	0,00	-1,27
32	-2,93	-0,70	0,00	-1,36
16	-4,40	-1,12	0,00	-0,35
8	-8,16	-2,99	-1,06	0,00
4	-15,04	-7,45	-3,65	0,00
2	-25,21	-14,74	-8,24	0,00
1	-39,15	-25,37	-15,89	0,00

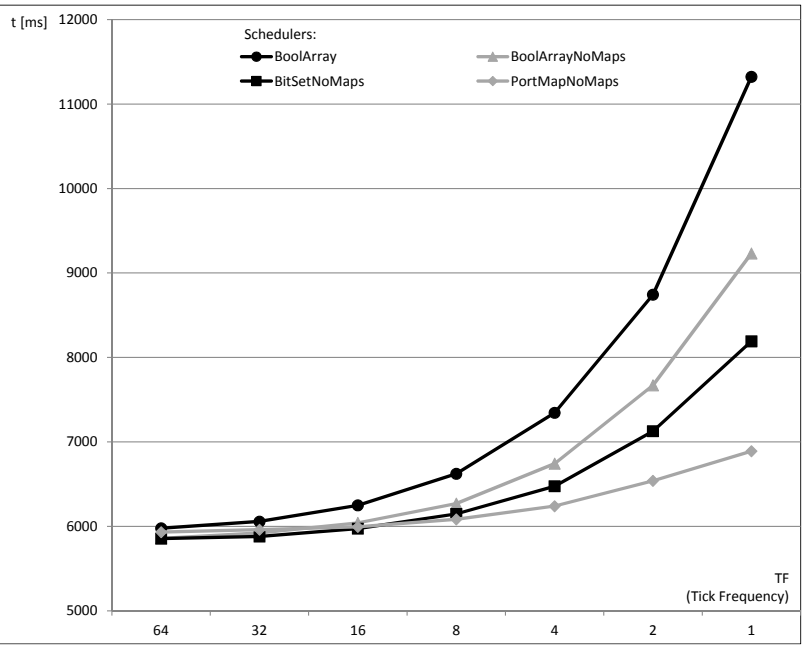


Figure 4.31: Scheduler comparison results of the average execution time in milliseconds for component LoadTest_100_8. The tick frequency of the distinct scenarios is ranging from 64 to one.

4.5.6 Discussion of the Results

Summing up, the following results are observed:

1. Generic schedulers that use a `HashMap` to map component instances to their tickfree ports are slower than their related scheduler variant that uses an `ArrayList` for this purpose.
2. `PortMap` based schedulers are the fastest in both scheduling scenarios for tick frequencies below 16 and are only slightly slower in the second setup for higher TFs than the `BitSet` based scheduler.
3. Generated schedulers that only manage a single component instance are faster than generic schedulers that have to manage multiple component instances.

As expected, result 1 is explained by the overhead needed to compute hash values for the components to identify the related tickfree data structure in the HashMap. Thus, the operation **resolveTickfree** (cf. Section 4.5.2) can be performed faster by using an `ArrayList` instead of a `HashMap` and address components with an unique index.

Result 2 is explained with a detailed look at the implementation of the scheduler and how the particular tickfree data structures need to be processed. In the first setup, BoolArray based schedulers have not been substantially slower than schedulers based on a PortMap. This is caused by the implementation of the operation **isPortTickfree** which simply checks the position determined by the port in the array. The operation **markPortBlocked** is implemented by simply switching the array position, again determined by the port, to false. Both of these array

operations are pretty fast. Nevertheless, in the second scenario, BoolArray based schedulers decrease in performance compared to PortMap based schedulers. This is explained by analyzing the corresponding implementations of operation **areAllPortsBlocked**. For a Boolean array it has to be checked whether all positions in the array contain the value false. This operation has a complexity of O(n), where n is the amount of scheduled ports. In contrast, a PortMap uses a Boolean array for the first two operations, but implements this operation by managing an integer variable that counts the amount of `false` values in the array. This way, the implementation of the last operation simply checks if this variable has the value zero, which results in a complexity of O(1). Thus, this measurable difference of 25,37 % between these two schedulers for TF 1 (only ticks) is explainable by the fact that in the second scenario 50 times more ports per component instance are involved than in the first scenario. Thus, operation **areAllPortsBlocked** is executed more often.

In both scenarios, BitSet based schedulers are slightly faster or slightly slower than PortMap based schedulers for tick frequencies of 64, 32, 16, and 8. With increased $\surd$ frequency, the performance decreases compared to PortMap schedulers. This effect indicates, that the operation **Is port tickfree** is realized similar performant, but the operations performed while scheduling ticks, **markPortBlocked**, **areAllPortsBlocked**, and **reorganize**, are realized less performant. Since all needed methods in the BitSet as well as the PortMap data structure have a constant complexity, these methods have to be analysed in detail.

Table 4.32 depicts how scheduling operations are realized with the respective data structure and how many atomic Java operations (e.g., comparison, array access, bit shift, assignments) are involved. For example operation **markPortBlocked** is realized with a BitSet by first checking whether the port is tickfree (`get(ID)`) and then marking it as blocked `clear(ID)`, which results in 24 Java operations. In the bottom row the operations needed to schedule the end of a time interval on n ports are summed up. Therefore, n ports have to be marked as blocked (n * M), n times it is checked, whether all ports are blocked (n * A), and finally the tickfree data structure needs to be reorganized (R). However, operation A is negligible, since it has the same complexity in both realizations. Thus, in total 40 atomic Java operations are executed for each port if a BitSet is used to manage tickfree ports. 10 atomic operations are executed if a PortMap is used. However, the PortMap scheduler is not four times faster than the BitMap scheduler since different Java operations have different execution time. Nevertheless, this comparison explains the measured performance advantage of PortMap based schedulers.

Finally, result 3 is explained by the advantage of directly integrating the tickfree data structure into the scheduler. Thus, it is immediately available and must not be resolved. The effect, that the internal scheduler is slightly slower than the external one is explained by a) that there are more method calls between port and its scheduler than between component and its scheduler and b) the external scheduler is a flat class without inheritance hierarchy. In contrast, the internal scheduler, which is integrated into the abstract implementation of the component, has a more complex inheritance hierarchy. Therefore, Java has to dynamically look up the location of the scheduler implementation within this hierarchy which renders the internal scheduler to be slightly slower.

Summing up, the PortMapNoMaps scheduler performs as the fastest generic scheduler variant in the described test setup. Therefore, it is selected as the default scheduler produced by the `SchedulerFactory`.

Operation	BitSet	PortMap
(I) isPortTickfree	get(ID) = 8	isTickfree(ID) = 1
(A) areAllPortsBlocked	isEmpty() = 1	allPortsBlocked() = 1
(M) markPortBlocked	get(ID) + clear(ID) = 8 + 16 = 24	setPortBlocked() = 5
(R) reorganize	n * set(ID) = n * 16	n * setPortTickfree(ID) = n * 5
$\surd$ scheduling on n ports: n * M + R	n * 24 + n * 16 = 40 * n	n * 5 + n * 5 = 10 * n

Table 4.32: Amount of Java operations needed to realize scheduling operations with a BitSet and a PortMap based scheduler.

It is possible to slightly improve the scheduling performance by using generated schedulers. Internal generated schedulers, however, have the drawback of their hard-wired integration into the component implementation. Thus, they cannot be replaced with other scheduler instances, e.g., for testing. On the other hand, a small memory overhead is produced by external generated schedulers since each scheduler is instantiated as a dedicated object. To avoid hard-wired direct integration by the generator, further handwritten component factory code is needed which assigns a scheduler instance to every instantiated component.

4.6 Technical Design Decisions

Several design decisions lead to the current design and implementation of the MontiArc simulation RTE and its corresponding scheduling. These are discussed in the following.

Separation of Timing and Simulation Properties into Different RTE Interfaces

In the MontiArc RTE, timing properties of components are encapsulated in the Java interface `ITimedComponent` and simulation specific information are encapsulated in the Java interfaces `ISimComponent`, `IOutSimPort`, and `IInSimPort` (see Figure 4.10, 4.11).

The idea behind this separation interfaces is first to encapsulate simulation specific methods away from the end user. If, e.g., the behavior of an atomic component is to be implemented, only user specific methods of ports (`send()`, `accept()`) are visible to the end user. Second, simulation components that represent untimed components do not implement the Java interface `ITimedComponent` (see Section 5.4.1). In this way, time related methods (`timeStep()` and `getLocalTime()`) are not available in untimed component implementations. Thus, the user, who implements the behavior of an untimed atomic component, is not able to directly acquire information about the simulated time.

Shared Port Objects

In the simulation, no distinct objects are created for outgoing ports and connectors. Connections are implicitly drawn by reusing the connected incoming port object as outgoing port. This is inspired by the FOCUS communication property of instantaneous transmission [BS01] and by requirement *SRQ10*, since additionally two simulation objects per connection are saved. If a special communication property like delay or message loss is to be simulated, this can be achieved by, e.g., using convenient library components that emulate the desired communication property (see Section 6.8 on page 217).

Shared Message Objects

MontiArc ports can have complex data types. Conceptually, if a component sends a complex object to a distributed receiver, it is first serialized and then deserialized. Thus, a new object is created at the receiver and neither the sender nor the receiver have the possibility to modify the message object of the other communication partner. To avoid the expensive serialization process in the simulation, only the pointer to the message object is transmitted to the receiver. Admittedly, if both components store these objects, they are able to manipulate the state space of the other component. This can be avoided by serializing the transmitted object or by creating a deep clone. However, the resulting overhead would slow down the simulation which contradicts with requirement *SRQ10*. Further, if only immutable objects are transmitted, the discussed problem does not occur either.

Incoming Port Driven Scheduling

Scheduling can be either incoming or outgoing port driven. Either incoming ports notify the scheduler that they received a message or tick and the scheduler controls the propagation to the component. Or outgoing ports, which want to transmit a message, registers at the scheduler, that then controls the message transmission to the connected incoming ports. The decision to schedule events on incoming ports is based on the following reasons:

a) All incoming ports of a component are needed to synchronize time intervals. If the scheduler would schedule outgoing ports, time interval synchronisation and the decision, whether a message can be delivered to the component of an incoming port, have to be handled by the component itself. As discussed in Section 4.5.1, this leads to additional overhead that slows down the simulation.
b) A message received on an incoming port corresponds to an event which is rendered for and delivered to the corresponding component of the port. This way, event target (component) and event trigger (port) belong to each other. If sending of messages on outgoing ports would have been triggered by the scheduler, event and event target belong to different components. Thus, scheduling incoming ports is most intuitive for FOCUS based simulations.
c) Scheduling incoming ports fits well to the “fire-and-forget” semantics of asynchronous communication. Here, senders simply emit messages and continue with own computations. The receiver has to buffer received messages until it is ready to process them.

d) More than one receiver can be connected to an outgoing port. Scheduling such an outgoing port would lead to the simultaneous creation of events on distinct connected components. But according to a), the scheduler is not directly able to decide whether these events can be delivered to the connected components. In the current implementation, messages that are sent from an outgoing port to multiple incoming ports are immediately transmitted to all connected incoming ports. The concrete events are then created individually by the scheduler when each connected port is scheduled.

Chapter 5

Technical Realization of MontiArc

The MontiArc language presented in Chapter 3 is realized using the MontiCore language workbench. Several transformations, analyses, and generation steps are needed to generate MontiArc simulation components (see Chapter 4). The language processing activities and corresponding implementations are presented in this chapter. They are designed to implement modular language processing tools which can be reused in extensions of MontiArc (see *RQ2*).

At first, MontiArc's main tooling is presented and an overview of the model processing steps is given. Section 5.2 documents MontiArc's symbol table, an important tool for an extendable language [Völ11], which is used for model analyses. The model transformations, that are discussed in Section 5.3, are a preparatory task for context condition checks as well as code generation. The latter is presented in Section 5.4. The mechanisms to implement the behavior of atomic components are presented in Section 5.5. Optimizations of the generator as well as MontiArc's runtime environment (RTE) are outlined in Section 5.6. The chapter ends with a presentation of the available MontiArc end user tools Section 5.7.

5.1 Model Processing

The MontiArc language and its corresponding tools are developed using MontiCore [GKR+08, Kra10]. Hence, the developed tools are based on MontiCore's DSLTool-Framework [Kra10, Chapter 9]. The most important classes are depicted in an abstracted class diagram given in Figure 5.1. A `DSLTool` is a tool that is able to process models of a certain `LanguageFamily`. Several `ExecutionUnits` are registered in a language family and can be accessed via a unique name. The most important `ExecutionUnit` is a `DSLWorkflow` which encapsulates an algorithm to process a model. Concrete algorithms are implemented as subclasses of `DSLWorkflow`. Hence, the following phases of model processing are realized as workflows. An `ExecutionUnit` always operates on an instance of a `DSLRoot` that represents a concrete model at tool runtime. Beside other information about the current model, e.g., its file name, the `DSLRoot` stores the abstract syntax tree (AST) of the model. A concrete `DSLRoot` subclass is generated by MontiCore for each language.

Workflows are executed in two different model processing phases. Workflows registered for the *synthesis phase* are executed by a `DSLTool` exclusively for all models that have been directly passed to the tool. If a workflow is registered for the *analysis phase*, it is additionally executed on models that are referenced by a processed model. This way, the execution of workflows can be controlled more fine grained.

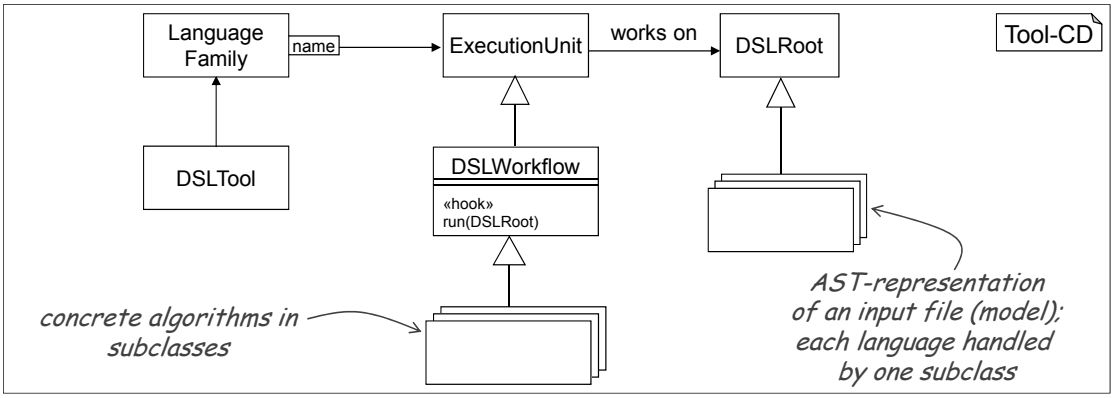


Figure 5.1: Important classes of the MontiCore DSLTool-Framework according to [Kra10].

MontiCore supports several language reuse mechanisms to integrate independently developed languages (cf. Section 3.4.1 on page 50). Regarding model processing, these mechanisms can be reduced to two cases that allow a seamless integration within a concrete language like MontiArc. According to Schindler [Sch12, Section 7.1], this integration is performed on two layers that arise from the structure and composition of the used grammars.

1. **Composition by language embedding:** Languages or parts of a language can be reused within another language. This allows to embed Java or OCL/P [Rum11, Chapter 3] as a constraint definition language into MontiArc.
2. **Composition by reference:** Models reference other models of the same or another language. The referenced model is not directly contained in the AST of a language but is stored as a reference (e.g., a qualified name). MontiArc uses composition by reference to reference data types of ports defined in Java or in class diagrams (CDs).

The most important classes of the MontiCore framework that technically support these two kinds of integration are depicted in Figure 5.2. Composition by reference is realized by a `LanguageFamily` which contains a set of `ModelingLanguages`. According to the technical configuration of the language family (see Section 5.2), the contained languages are allowed to reference each other. A modeling language contains MontiCore-specific information like the file extension of the processable models or available workflows. A `ModelingLanguage` has an `ILanguage` that represents the concrete used language. According to the realized composite pattern [GHJV95], it can be either a `LanguageComponent` that represents a single language (or a part of a language) or a `CompositeLanguage`. The latter realizes composition by language embedding and contains all composed `ILanguages` and the configuration to technically integrate the composed languages (see Section 5.2).

Figure 5.3 shows the concrete object graph that is instantiated by the MontiArc `DSLTool`. The instantiated language family contains an instance of `JavaLanguage`, `CDLanguage`, and `MontiArcLanguage`. This way a MontiArc tool can process Java classes, CDs, and MontiArc models that contain references to types defined in Java or a CD (composition by reference). The linked instance of `CompositeMontiArcOCLAndJavaLanguage` (this class

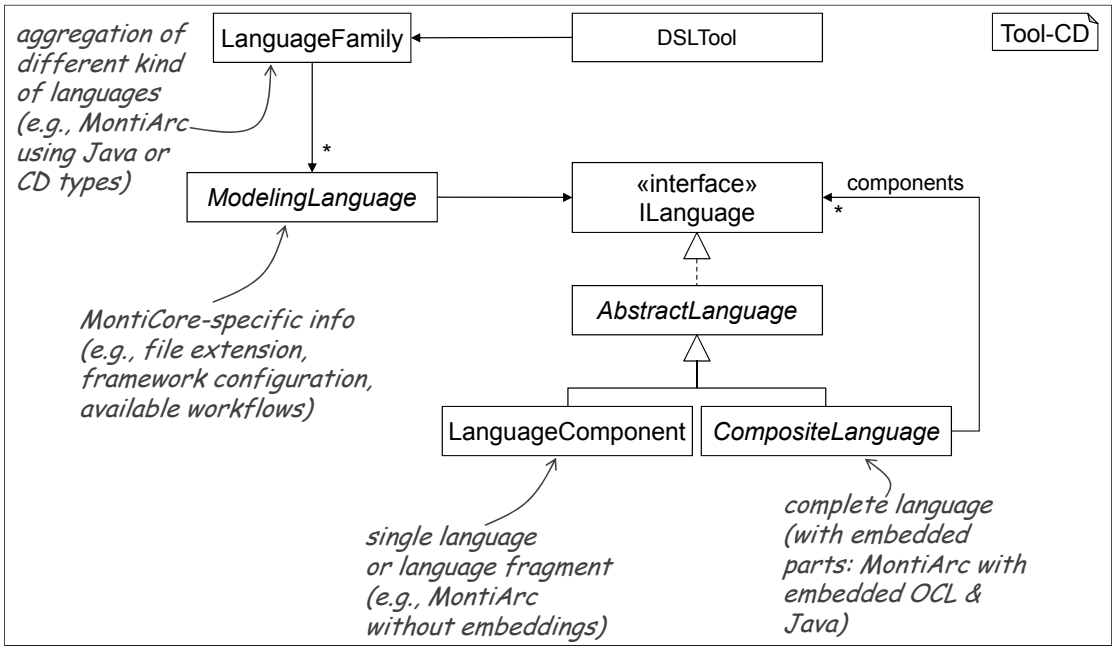


Figure 5.2: Technical realization of MontiCore's language composition mechanisms according to [Völ11, Sch12].

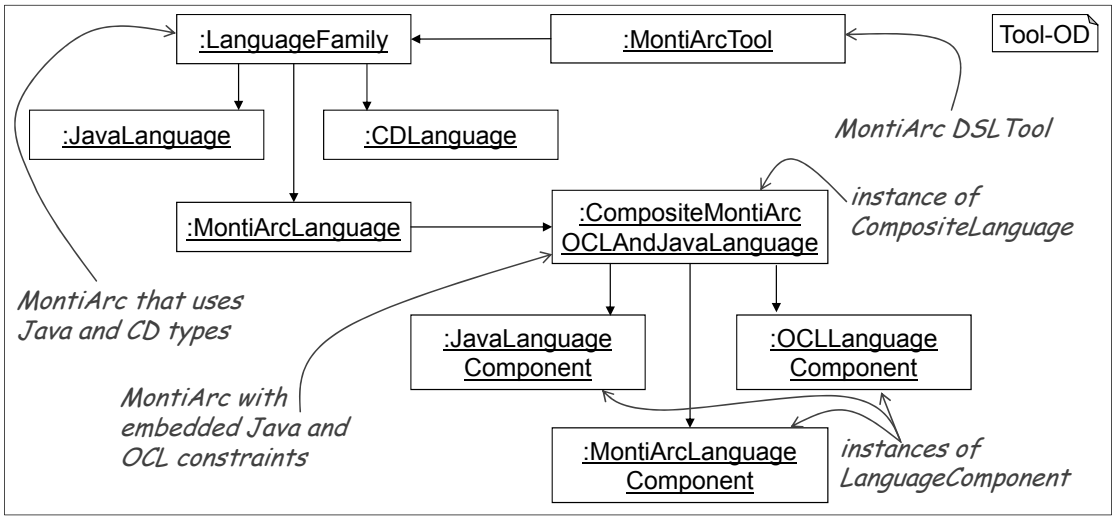


Figure 5.3: Object graph of the MontiArc DSLTool with the technical languages it uses.

extends `CompositeLanguage`) allows to process MontiArc models with embedded Java or OCL constraints. Hence, it links an instance of the language components for Java, MontiArc, and OCL (composition by language embedding).

An overview of the workflows used to process MontiArc models is given in the activity di-

agram (AD) depicted in Figure 5.4. The workflows that are executed in the analysis phase are located in the top right part of the figure, the synthesis workflows are located right beyond the dashed line. Activities and artifacts located at the left belong to the tool environment, elements at the right are performed within the MontiArc DSLTool.

The analyse workflows are executed on files which are directly passed to the tool. If no exported symbol table of the referenced model exists yet, these workflows are also executed on referenced models. The analysis workflows are:

1. **Parsing:** The parsing workflow takes a MontiArc component file as input and creates a corresponding AST. It is instantiated using the AST classes which are generated by MontiCore. If parsing fails due to an invalid syntax, the MontiArc tool stops with an error. This workflow is registered with id `parse`.
2. **Create Symbol Table:** This workflow traverses the AST using a visitor [GHJV95] and creates a parallel data structure called symbol table. The structure of MontiArc's symbol table and the workflow to create it is presented in detail in Section 5.2. This workflow is registered with id `init`.

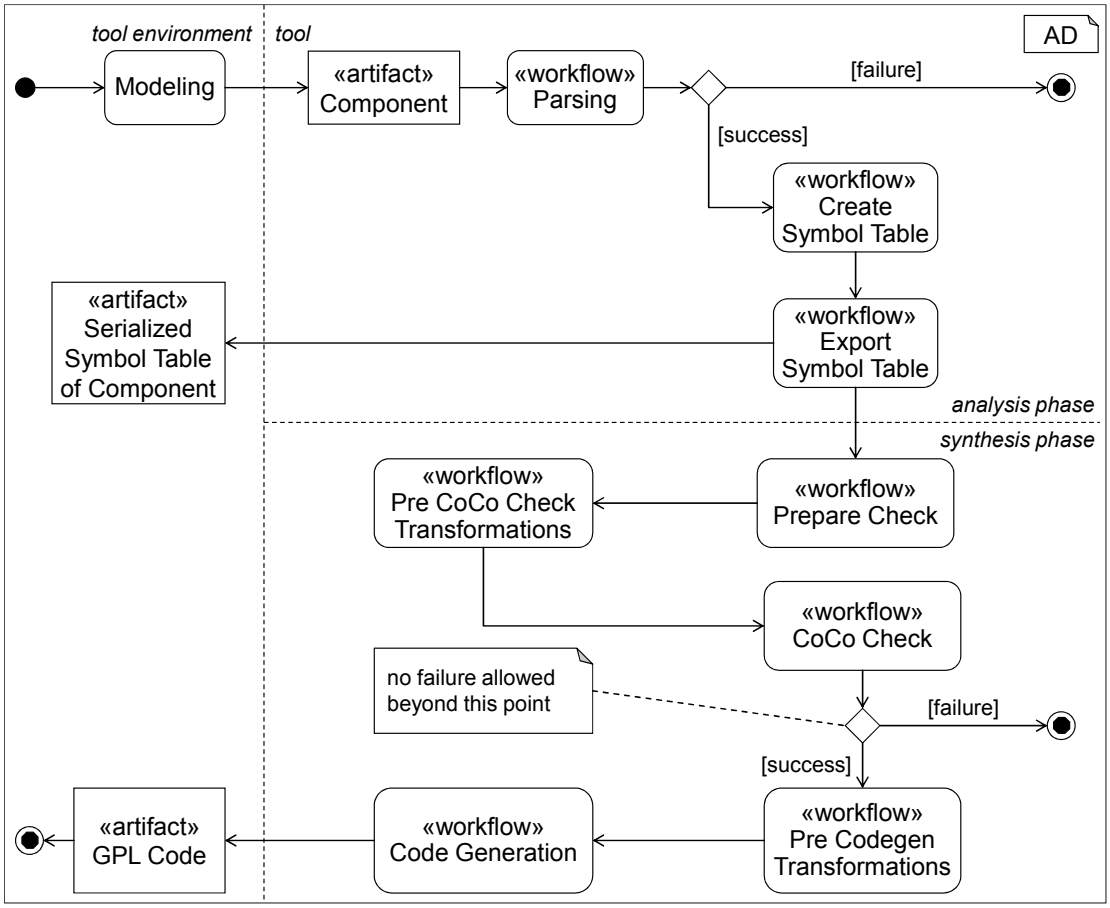


Figure 5.4: Overview of MontiArc workflows and produced artifacts.

3. **Export Symbol Table:** This workflow serializes the symbol table to files. If a referenced model has to be loaded, e.g., to check its port types, its serialized symbol table is loaded instead of its original model file (see Section 5.2). This workflow is registered with id `createExported`.

After the analysis phase has finished successfully, the workflows of the synthesis phase are executed on models that are directly passed to the MontiArc tool. The synthesis workflows are:

1. **Prepare Check:** Further steps needed to prepare the symbol table. This workflow is registered with id `prepareCheck`.
2. **Pre CoCo Check Transformations:** A set of model transformations that expand comfort functions such as `autoconnect` (see Section 3.3) into regular models. They mostly prepare the AST to simplify and unify context condition checks. The used transformations are described in detail in Section 5.3.1. This workflow is registered with id `preCheck-Transformation`.
3. **CoCo Check:** This workflow checks the context conditions discussed in Section 3.5. The framework used to implement and check the realized conditions is explained in [Sch12, Section 7.3]. If checking of context conditions fails, the whole process is aborted and the following steps are omitted. This workflow is registered with id `check`.
4. **Pre Codegen Transformations:** A set of transformations executed before the code generation starts. Mainly unqualified data types are replaced by their qualified version in the AST. These transformations are described in Section 5.3.2. This workflow is registered with id `preCodegenTransformation`.
5. **Code Generation:** A workflow that uses Freemarker [www13b] templates to transform the AST of a component into general purpose language (GPL) code. The used code generation framework is explained in [Sch12, Section 7.4]. The performed model-to-code transformations are explained in detail for all model elements in Section 5.4.

Please note that these workflows represent the default model processing as performed by the MontiArc Maven Plugin (cf. Section 5.7.2). It is always possible to omit certain workflows, e.g., to solely validate the processed models and skip code generation, or to register other workflows. How to use the MontiArc Tool is described in Section 5.7.1. A method for the extension of the MontiArc tooling is presented in Section 7.1.

5.2 Symbol Table

A symbol table is an important data structure that enables compositional development of languages [Völ11, HLMSN+15]. Consequently, it is an essential infrastructure for an extendable modeling language such as MontiArc. Beside the infrastructural support for language composition, the MontiArc symbol table further allows an efficient implementation of MontiArc context conditions (see Section 3.5) and additionally supports the implementation of transformations (see Section 5.3) as well as code generation for MontiArc simulations (see Section 5.4).

After summing up the foundations and the construction process of the symbol table in the following two subsections, the namespace hierarchy (Section 5.2.3), the class structure (Section 5.2.4), and the interface of a MontiArc model (Section 5.2.5) are explained.

5.2.1 Foundations

According to Völkel [Völ11], a *symbol table* is a data structure to store and to resolve identifiers within a language. Its core task is to resolve names to gather further associated information such as data types or signatures. The MontiArc symbol table uses the symbol table framework presented in [Völ11, Chapter 7] that supports compositional development of languages. Hence, in the following it is adhered to the definition of the terms given in [Völ11].

A fundamental concept of a symbol table are *entries*. An entry has a specific *entry kind* that reflects the model element kind (e.g., a class or a method) it represents. Entries have a resolvable name and further associated kind-specific information. An entry can be in one of three possible states: *unqualified*, *qualified*, or *full*. An unqualified entry is in its initial state and only the unqualified name of the represented element is known. In the qualified state, the full qualified name of the represented element is known. This especially occurs, if the represented element is not defined in the current model. If an entry is in the full state, it additionally contains all kind-specific information about the represented model element. Entries are automatically transferred from the unqualified state to the qualified state during the entry qualification phase (see above). The transition between the qualified and the full state has to be triggered manually, when detailed information about the represented model element of an entry is needed. Please note that these information can contain further entries in a qualified state.

An entry is visible within a certain *scope*. A scope is a part of a model, in which the model element that is represented by the entry can be referenced by its name. [Völ11] implements the scope concept with hierarchical *namespaces* that manage the entries of a symbol table and allow hiding of names (e.g., a local variable name in a Java method can hide a global field name within the same class). A namespace represents a segment of a model and is able to import entries from another namespace (e.g., from a hierarchical parent namespace). These imported entries may be hidden by local entries with the same name. As depicted in Figure 5.5, a namespace has four different kinds of symbol tables that influence this import mechanism.

1. The `encapsulated` symbol table contains elements that are solely visible within the namespace.
2. The `exported` symbol tables contain elements that are propagated to the environment of a model.
3. The `imported` symbol tables contain elements defined within another namespace (e.g.,

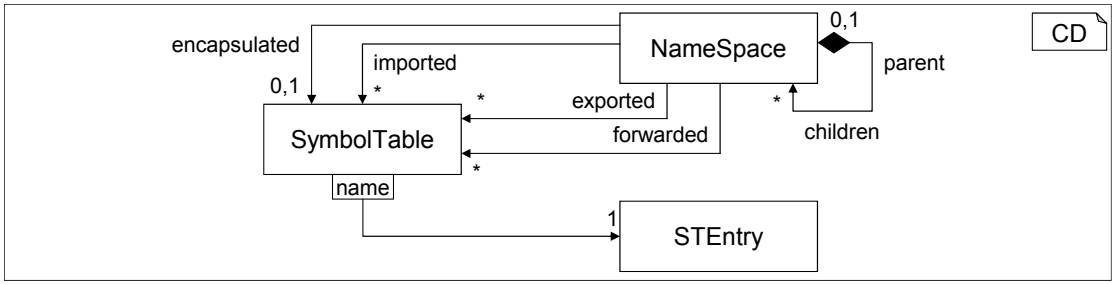


Figure 5.5: Relation of namespaces, symbol tables, and entries (according to [Völ11]).

elements of the exported symbol table of the parent namespace) that are imported into the current namespace.
4. The `forwarded` symbol tables contain elements imported from another namespace and exported to the environment of a model.

The entries within a `SymbolTable` always have a unique name.

5.2.2 Symbol Table Construction

The MontiCore DSLTool-Framework already provides components needed to construct a symbol table based on the method presented in [Völ11]. These components are registered at an `ILanguage` and handle the symbol table of either a concrete language component or a composite language (see Figure 5.2). The most important symbol table components are depicted in Figure 5.6. These are:

1. `ILanguage`: All technical components of a symbol table are registered at an `ILanguage`. As previously discussed in Section 5.1, concrete `ILanguages` have to be developed for a modeling language. Depending on the used composition mechanism, this can be either a language component or a composite language. Beside the aggregation of technical symbol table components, an `ILanguage` provides the information which AST nodes of a modeling language create a new namespace.
2. `ConcreteASTAndNameSpaceVisitor`: A special visitor (see [GHJV95]) used to create the symbol table for a model. A concrete subclass has to be hand-written for a concrete language that has to visit all relevant AST nodes and has to create symbol table entries for the visited nodes. The class `ConcreteASTAndNameSpaceVisitor` additionally provides some helper methods to, e.g., get the namespace of current AST nodes and thus the suitable symbol tables in which the created entries should be added.
3. `IQualifierClient`: Qualifier clients are responsible for automatically transfering entries of a certain kind from their initial unqualified state into a qualified state. Concrete qualifier clients have to be created for custom entries that can be referenced in a model. For example, data type entries are most often referenced by other models.
4. `STEntryDeserializer`: Entry deserializers are responsible for loading serialized entry interfaces of referenced models. They are automatically called if an entry should be transfered from a qualified to a full state. The deserializer then loads the serialized interface and extracts the contained information into an entry object. An `STEntryDeserializer` has to be registered for every used entry kind.
5. `IInheritedEntriesCalculatorClient`: Inherited entry calculators compute if an entry is hidden by another entry. This allows to shadow names of outer namespaces with names of inner namespaces. One concrete inherited entry calculator has to be developed for each language that supports this hiding mechanism.
6. `IResolverClient`: Resolver clients are responsible for resolving entries by their name in the created namespace hierarchy. A concrete resolver client has to be written for every entry of a modeling language.
7. `AContextCondition`: Registered context conditions that are executed for specific model elements of the language.

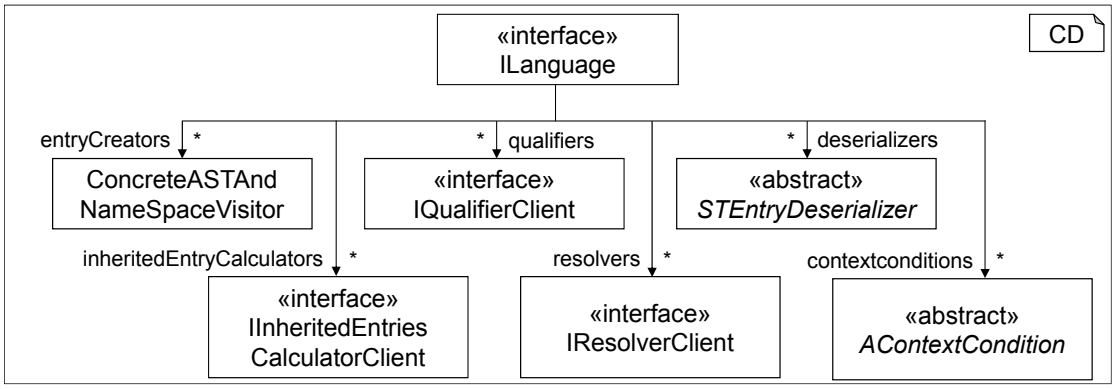


Figure 5.6: Most important technical symbol table components (according to [Völ11]).

Please note that the MontiCore DSLTool-Framework already offers some default implementations for these components that, e.g., provide default qualification or resolving mechanisms. Hence, not all symbol table components for a modeling language have to be developed from scratch.

The symbol table of a model is constructed in nine phases. These phases are either bound to different workflows of a tool execution or have to be executed explicitly (see Figure 5.4). These phases automatically use the registered symbol table components to compute and serialize the symbol table of a model.

Steps performed in the **Create Symbol Table** workflow in the analysis phase of a tool execution are:

1. *Namespace setup:* Creates a namespace object for the AST nodes denoted by the `ILanguage`. The created namespace objects have a hierarchical structure that corresponds to the hierarchy of the AST. The relation between the MontiArc AST and namespaces in the MontiArc symbol table is explained in Section 5.2.3. Additionally the relation between AST nodes and namespaces is stored.
2. *Creating and registering entries:* Entries are instantiated and linked by a concrete `ConcreteASTAndNameSpaceVisitor`. Furthermore, the created entries are stored in the symbol tables of the corresponding namespaces. The created structure of the MontiArc symbol table is described in Section 5.2.4.
3. *Connecting the namespaces (optional):* The different aforementioned symbol tables of the already hierarchical connected namespaces are organized in this step (imported, exported, encapsulated, and forwarded). Furthermore, hidden entries are marked.
4. *Entry qualification:* Unqualified entries of the processed models are qualified using the registered qualifier clients.

The following step is executed in the **Export Symbol Table** workflow:

5. *Serialize model interfaces:* Since all entries of a model are (at least) in a qualified state, the interface of the model can now be exported. A *model interface* serves as an interface between different models. It contains the entries which are exported from a model and can be imported into the namespace of another model. The various model interfaces of

the MontiArc symbol table are described in Section 5.2.5. Please note that in contrast to the grammar based serialization proposed in [Völ11], MontiArc uses an annotation based generic approach provided by the MontiCore DSLTool-Framework. Annotated fields of an entry class are automatically serialized.

The following steps are executed in the **Prepare Check** workflow during the synthesis phase of a tool execution:

6. *Import symbol tables:* In this phase, entries from other referenced models are imported into the namespaces of the current model. This way, e.g., entries from a superclass are visible in a subclass. Please note that the imported entries are taken from the serialized model interface of the referenced model (see step 5).
7. *Connecting the namespaces (optional):* The different symbol tables have to be reorganized due to the import of further symbol tables (step 6). Again, it has to be checked whether newly imported entries are hidden by already existing entries. After this step the symbol table construction is finished.
8. *Set resolver clients:* The registered resolver clients of the current language setup are registered at a central `Resolver` component.

The last step is triggered manually by calling the `loadFullVersion` method of an entry:

9. *Load qualified entries:* In this phase, entries are transfered from their qualified state into a full state on demand. The framework loads the model interface of the qualified entry, extracts all contained information into a full entry, and replaces the qualified entry with the full entry. Please note that the registered entry deserializers are used to create entries from a model interface.

5.2.3 MontiArc Symbol Table: Namespace Hierarchy

Four different MontiArc AST node types open a new namespace in the namespace hierarchy. With the exception of the first node, namespaces are opened and closed by curly brackets in the concrete syntax of a MontiArc model. The corresponding AST nodes are:

1. `CompilationUnit`: The MontiArc compilation unit opens a new namespace. It is inserted into the package namespace that is automatically created by MontiCore's symbol table framework.
2. `ArcComponent`: Each component definition opens a new namespace. The namespace of the top level component is a direct child of the compilation unit namespace. Namespaces of inner component definitions are directly added as child to the namespace of the component definition in which the inner component is defined.
3. `MontiArcInvariant`: A constraint definition opens a new namespace. These namespaces are a child of a parent component namespace. Please note that, depending on the concretely used constraint description language, this namespace can have further child namespaces which are created by the symbol table of the embedded language.
4. `ArcComponentImplementation`: A component implementation opens a new namespace. These namespaces are directly added to the namespace of the parent component.

Please note, the depth of the namespace hierarchy of MontiArc models is unrestricted since component definitions can recursively contain further inner component definitions. This recursive

containment relation is directly reflected by the corresponding namespaces.

The corresponding namespace hierarchy of a MontiArc model is exemplary depicted in Figure 5.7. On the left-hand side, component definition `LightCtrl` is depicted. The right-hand side shows an object diagram (OD) of the resulting namespaces. The compilation unit (the complete MontiArc model file) corresponds to the namespace `compilationUnit`. Namespace `lightCtrlComponent` is its direct child that itself corresponds to the component definition within the compilation unit. It has two child namespaces: `fooConstraint` and `arbiterComponent`. The former corresponds to the constraint definition `foo` that is contained in component `LightCtrl`, while the latter corresponds to the inner component definition `Arbiter`. The contained implementation corresponds to namespace `implementation` which is a child of namespace `arbiterComponent`.

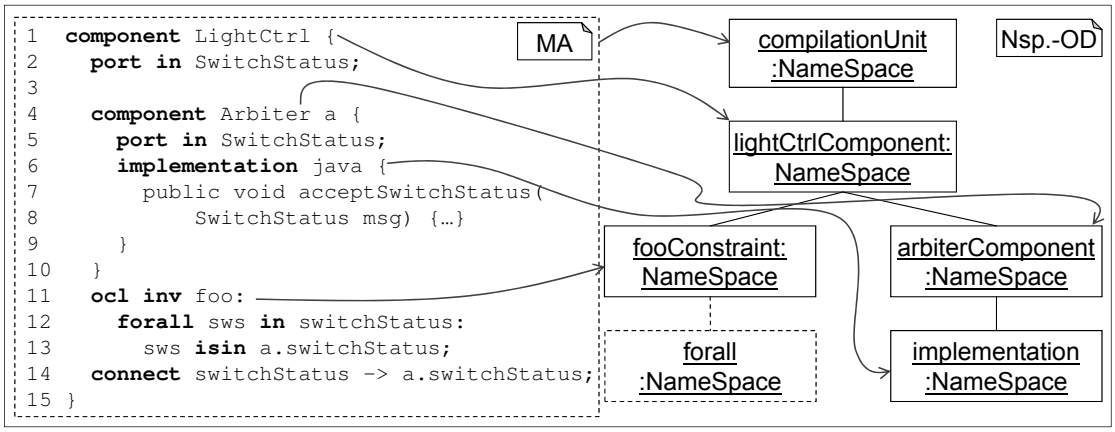


Figure 5.7: Namespace hierarchy of component `LightCtrl`.

No further namespaces are created for the other depicted MontiArc elements such as ports or connectors. The entries, that represent these elements, are contained in the corresponding namespace, while entries of elements that open a new namespace are contained in the parent namespace. This is demonstrated in the following:

- Namespace `compilationUnit` contains the component entry `LightCtrl`.
- Namespace `lightCtrlComponent` (ll. 1 – 15) contains:
  1. port entry `switchStatus` (l. 2),
  2. component entry `Arbiter` (ll. 4 – 10),
  3. subcomponent entry `a` (l. 4),
  4. constraint entry `foo` (l. 11), and
  5. connector entry `a.switchStatus` (l. 14).
- Namespace `fooConstraint` (ll. 11 – 13) does not contain entries since the OCL symbol table does not create entries for `forall` statements. Therefore, it only holds its child `forall` namespace.
- Namespace `forall` (ll. 12f) contains an entry for OCL variable `sws` (l. 12) which is used as a loop variable in the `forall` statement.
- Namespace `implementation` (ll. 6 – 9) contains an entry for method `acceptSwitch-`

`Status` (l. 7) provided by the Java symbol table. Please note, this method also opens a new namespace in the Java symbol table.

5.2.4 MontiArc Symbol Table: Structure

The entry classes of the MontiArc symbol table and their relations are depicted in Figure 5.8. Please note that all realized entry classes of the symbol table have an additional name attribute that is omitted in the following. A detailed description of the existing entry kinds is given in Table 5.9.

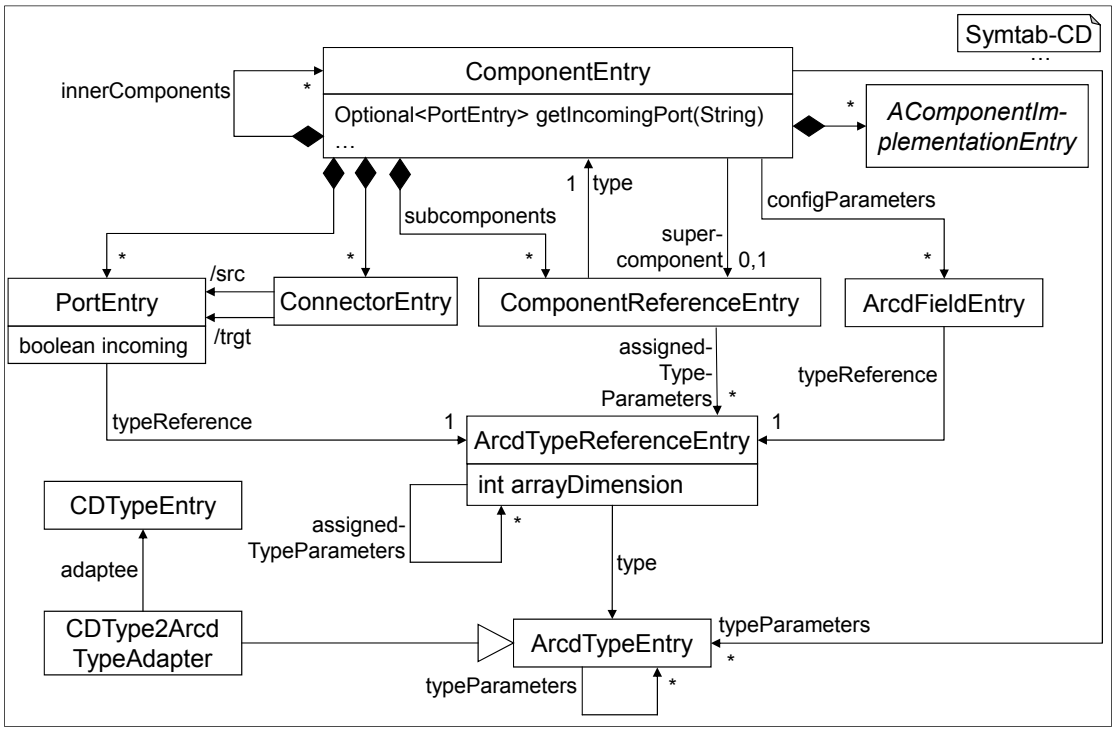


Figure 5.8: Entry classes of the MontiArc symbol table and their relations.

Entry Kind	Description
`ComponentEntry`	A `ComponentEntry` is created for each MontiArc component definition. A component entry consists of further entries that describe the component's interface and decomposition. The interface is given by a set of associated port entries. Optionally, a component entry has a `superComponent`. Configurable components further have configuration parameters (`configParameters`).

Table 5.9 continued on next page

Entry Kind	Description
ComponentEntry *(continued)*	Entries of generic components have type parameters (typeParameters) that represent the local definition of generic types within the component. The internal structure of a component is represented by inner component definitions (innerComponents) that are component entries, too. The internal decomposition of a component is represented by subcomponents, connector entries, and component implementations (AComponentImplementationEntry).
PortEntry	Port entries are either incoming or outgoing, stated by the value of the Boolean attribute incoming. A port also has a type that is stored using an ArcdTypeReferenceEntry.
ConnectorEntry	Connector entries represent connectors in the model which connect a source port (src) with a target port (trgt). Actually, connectors are not a candidate for an entry since a connector does not have a name. Nonetheless, connectors are implemented as entries because the underlying symbol table framework presented in [Völ11] realizes inheritance based on entries. Hence, connectors are inherited from a supercomponent. Additionally, this way connectors can be automatically resolved in the symbol table using the name of their target.
ComponentReferenceEntry	A component reference entry represents a reference to a component type. It is used to represent subcomponents as well as the reference to the type of a supercomponent (see above). The referenced component type is represented by the association type. If a generic component is referenced, assigned type parameters (see below) contain the concretely assigned type parameters. If a configurable component is referenced, the concrete arguments (configArguments) are represented by a ValueEntry.
ArcdFieldEntry	Field entries represent parameters of configurable components (association configParameters). Field entries reference a data type that is given by association typeReference.
ArcdTypeReferenceEntry	The MontiArc symbol table distinguishes between type definitions and a reference to a type definition. Data type definitions are represented via entry class ArcdTypeEntry, a reference to a data type is given by class ArcdTypeReferenceEntry. This is especially needed to support generic type parametrization. A data type definition can have type parameters (typeParameters) to store generic type definitions that are visible within the defined type.

Table 5.9 continued on next page

Entry Kind	Description
ArcdType-ReferenceEntry *(continued)*	If a generic type definition is referenced in another context, the generic type has to be assigned. This assignment is represented via the association assignedTypeParameters.
ArcdTypeEntry	Represents the definition of a data type. Generic data types have associated type parameters. Since MontiArc does not provide an own data type definition language, it adapts UML/P CDs and Java for this purpose. Therefore, the symbol table contains an adapter for each used type language which translates entries of the type language to ArcdTypeEntrys. This is exemplary depicted for CD types.
CDType2Arcd-TypeAdapter	Realizes the desired language aggregation of MontiArc and CDs. The adapter translates the associated adaptee CDTypeEntry to an ArcdTypeEntry. More details about this method are given in Section 7.3.2 and [Völ11, Section 6.3].

Table 5.9: Entry kinds of the MontiArc symbol table.

To ease finding elements within the symbol table, entry classes offer several helper methods which allow to search for specific elements by their name. This is indicated by the method getIncomingPort(...) that takes a name as argument and returns the corresponding port entry encapsulated within an Optional[1] if a port with the given name exists. If an incoming port with the given name does not exist, the caller is notified with the returned Optional.absent.

In the following the distinction between type and type reference is demonstrated by means of an example. Figure 5.10 depicts the simplified definition of Java type Set with its generic type parameter T in the top-left corner. In the MontiArc symbol table, these type definitions are represented by instances of ArcdTypeEntry. The object representing the local type definition of type parameter T is linked to the Set type definition using the typeParameters association (see Figure 5.8). A component definition Foo is depicted in the bottom-left corner. Its port s has the type Set parametrized with String. In the symbol table (right part of the Figure), this is realized with object Set which is an instance of ArcdTypeReferenceEntry. It is linked to the port object using the typeReference association. Generic parametrization of Set with type String is reflected by the assignedTypeParameters link to the object String that is also an instance of ArcdTypeReferenceEntry. Both type reference objects have a link to the objects that represent their referenced type definition (java.util.Set and java.lang.String). If the port object in the symbol table had been directly linked to the type definition, the information about the concrete generic parametrization would not have been available in the symbol table. For this reason, the MontiArc symbol table distinguishes between type definitions and a reference to a type definition.

[1] Optional is a class of the Guava library that is used for safely handling possibly not existing return values. See http://code.google.com/p/guava-libraries/ for more information about Guava. Since Java 1.8, a similar Optional implementation is also part of package java.util.

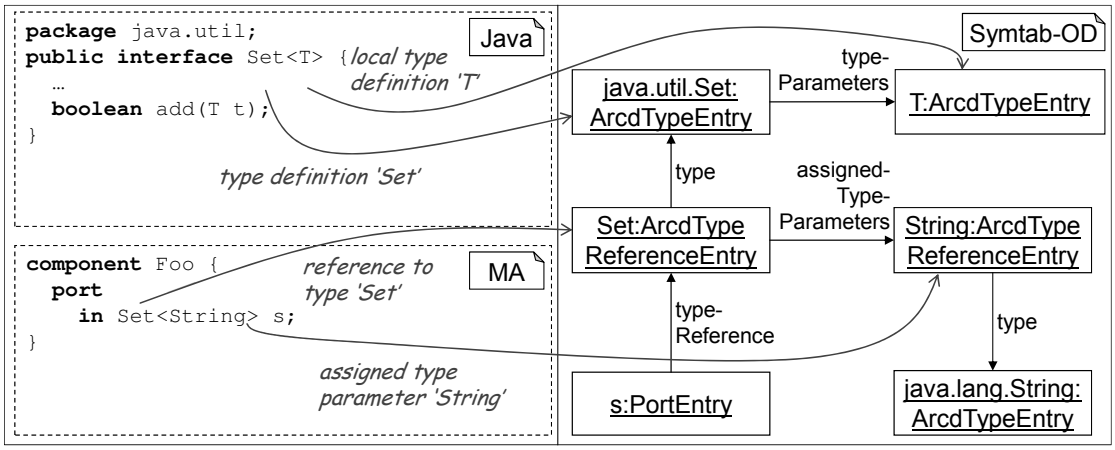


Figure 5.10: Relation between type definition and type reference manifested in the MontiArc symbol table.

5.2.5 MontiArc Symbol Table: Model Interfaces

The symbol table framework presented in [Völ11] supports the definition of different *model interface kinds*. This way, fine-grained visibility concepts like in Java can be realized. For every model interface kind, a dedicated interface with kind-specific model information is exported.

MontiArc defines two different model interface kinds: public and protected. The former only contains information about the component interfaces. It offers all information needed to instantiate a component definition using a subcomponent declaration and connect its ports with other subcomponents. The latter additionally contains information about the internal structure of a component and is used to realize component inheritance. Table 5.11 depicts which model elements are part of the public and protected model interface of the MontiArc symbol table.

Figure 5.12 demonstrates the difference between the public and the protected model interface of the MontiArc symbol table with an example. The top of the figure contains an excerpt of the component definition `LightCtrl`. It contains a port named `sStat` (l. 2) and the inner component definition `Arbiter` which is immediately instantiated to subcomponent `a` (ll. 4–6). The inner component definition has a port named `sStatInner`. The connector connects the depicted ports (l. 7).

The bottom-left part of the Figure depicts an OD that contains the objects of the public model interface. It can be seen that the public interface only contains the entry for the component definition `LightCtrl` and its incoming port `sStat`. The incoming port is linked with its type reference that itself is linked with its type definition (see above, omitted for reasons of space).

The bottom-right part of the Figure depicts the objects of the protected model interface of the same component. The gray objects and links are also part of the public model interface and are described above. Additionally, the component entry contains and entry for the inner component definition `Arbiter`, a subcomponent entry `a` which instantiates the inner component definition, and an entry for its incoming port `sStatInner`. The protected symbol table also contains a connector entry that connects the shown ports.

Language Element	Model Interface	
	Public	Protected
Top level component definitions	×	×
Ports of top level component	×	×
Configuration parameters of top level component	×	×
Generic type parameters of top level component	×	×
Subcomponents		×
Connectors		×
Inner component definitions		×
Ports of inner component definition		×
Configuration parameters of inner component definition		×
Generic type parameters of inner component definition		×

Table 5.11: Elements of the public and protected model interfaces.

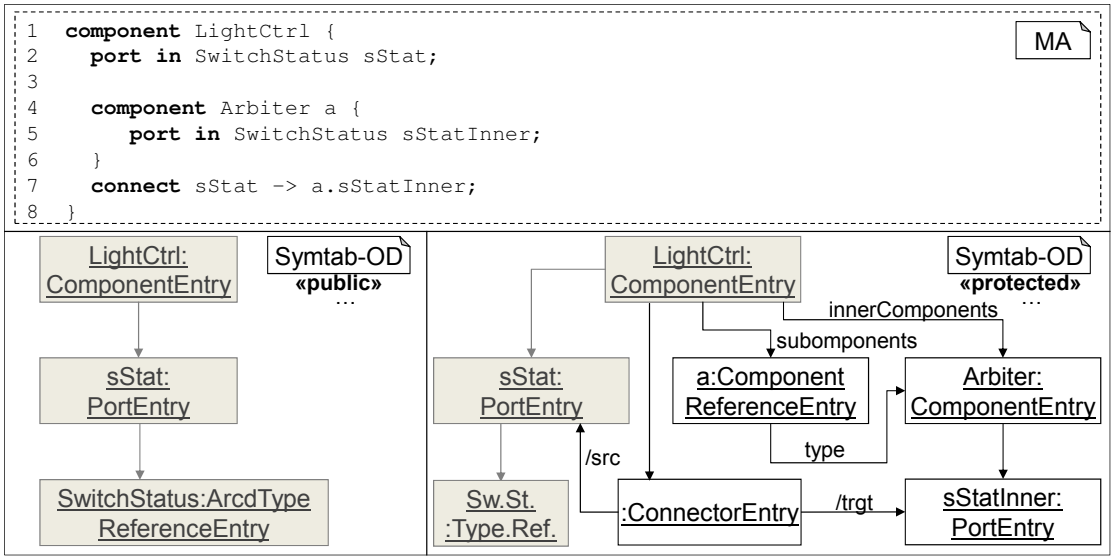


Figure 5.12: Difference between the public and the protected model interface.

Summing up, the gray objects and links shown in Figure 5.12 are publicly visible if component `LightCtrl` is instantiated as a subcomponent. The other objects and links are only visible if a component definition extends component `LightCtrl`.

5.3 Transformations

MontiArc provides two workflows for model transformations. As depicted in Figure 5.4, the first workflow `Pre CoCo Check Transformations` is executed before context conditions are checked, the second workflow `Pre Codegen Transformations` is executed between context condition checks and simulation code generation. In both workflows, transformations are registered which transform the AST and symbol table to ease further processing of the model. An overview of the realized transformations and the corresponding workflows is given in Table 5.13. The concrete transformations are discussed in Section 5.3.1 and Section 5.3.2. Section 5.3.3 explains how transformations are implemented in MontiArc. Please note that MontiArc's concrete syntax is used to depict the effect of the transformations, even though the discussed transformations only affect the model's AST and symbol table. Following the notation of [Rum12], the top part of the examples depicts the initial situation, the part below the horizontal line depicts the result of the transformation.

Transformation	**Workflow**
Instantiation of Named Inner Component Definitions	Pre CoCo Check Transformations
Qualify Subcomponent Connectors	Pre CoCo Check Transformations
Expand Autoconnect	Pre CoCo Check Transformations
Expand Autoinstantiate	Pre CoCo Check Transformations
Name Implicitly Named Subcomponents	Pre Codegen Transformations
Name Implicitly Named Ports	Pre Codegen Transformations
Expand Simple Connectors	Pre Codegen Transformations
Qualify all Types	Pre Codegen Transformations
Unconnected Incoming Ports of Subcomponents	Pre Codegen Transformations
Unconnected Outgoing Ports of Decomposed Components	Pre Codegen Transformations

Table 5.13: Overview of MontiArc transformations executed in the associated workflow.

5.3.1 Pre Context-Condition Transformations

The transformations described in this subsection are executed after the symbol table has been created and before context conditions are checked. Since the MontiArc language contains several shortcuts to ease textual modeling, e.g., auto connection of compatible ports, these transformations mostly map shortcuts to an equivalent form. This allows an unified checking of context conditions on an expanded model without repetitive resolving of implicit connections. The following transformations are executed before context conditions are checked.

Instantiation of Named Inner Component Definitions

MontiArc allows to define inner components similar to private inner classes in Java. To use an inner component in the context of a component, a subcomponent instance has to be created (see Section 3.4.3). If the optional `instanceName` is used for an inner component definition, this transformation automatically creates a subcomponent that instantiates the inner component definition.

This is demonstrated in Figure 5.14. The upper part of the figure depicts that component `LightCtrl` contains an inner component definition `AlarmCheck` which has the instance name `ac`. The lower part of the figure shows the result of the transformation. The instance name has been removed and a subcomponent declaration named `ac` has been created which instantiates the inner component definition.

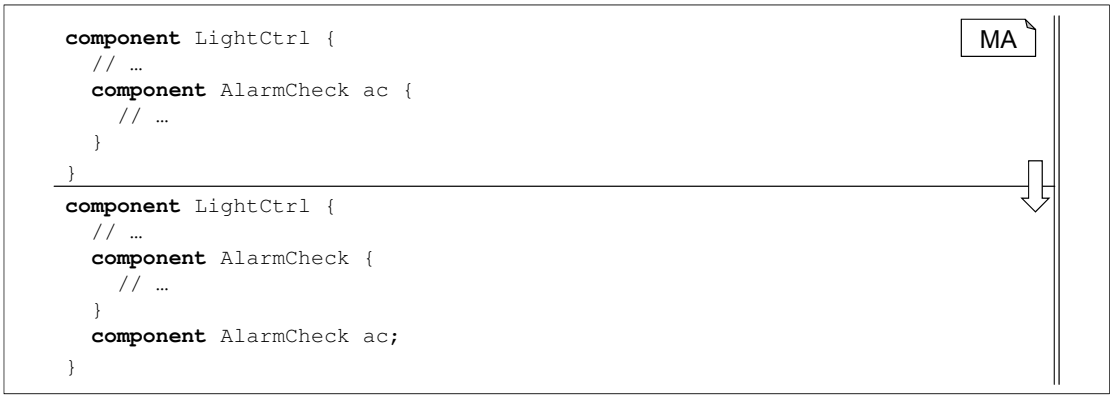


Figure 5.14: Pre CoCo Trafo: Instantiation of Named Inner Component Definitions.

Qualify Subcomponent Connectors

MontiArc supports the direct connection of subcomponents without specifying which concrete ports of the subcomponents have to be connected. If this short form is used, all type-compatible ports of the referenced subcomponents are connected.

Figure 5.15 demonstrates this with the aid of an example. Component `LightCtrl` contains an implicitly named subcomponent `arbiter` and a subcomponent `ac`. Both are directly connected with a connector. The top-right of the figure depicts the corresponding types of the subcomponents. Component `AlarmCheck` contains outgoing ports with type `BlinkRequest` and `String`, component `Arbiter` contains compatible incoming ports. The result of the transformation is depicted in the lower part of the figure. The two concrete connectors `ac.blink -> arbiter.br` and `ac.string -> arbiter.s` replace the underspecified connector `ac -> arbiter`. Please note that this transformation only connects ports which are type compatible and unique. If two or more ports from a receiver have a type that is compatible with the type of a sending port, these ports are not connected.

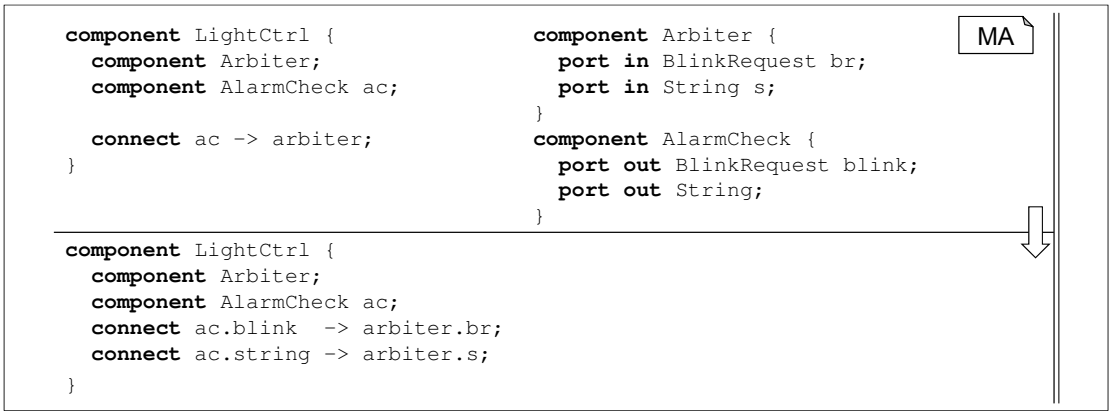


Figure 5.15: Pre CoCo Trafo: Qualify Subcomponent Connectors.

Expand Autoconnect

MontiArc provides two autoconnect strategies (see Section 3.3). The `port` connectors completion strategy creates connections for ports with the same unique name and compatible types that are not explicitly connected. The `type` connectors completion strategy connects unambiguous unconnected ports with compatible types disregarding port names. Both strategies create concrete connectors for yet unconnected ports that obey their matching criteria.

The steps to find matching port pairs are pretty similar. At first, yet unused sending and receiving ports from the current component and its subcomponents are collected. Then, it is iterated over both sets and the matching criteria is checked pairwise. If two ports fulfill the criteria, a connector candidate is created. If exactly one candidate is found for a receiver, the candidate is added to the AST and the symbol table. If more or none candidate is found, a warning is emitted.

Both strategies and their different behavior are demonstrated in Figure 5.16. In component `LightCtrl`, an `Arbiter` and an `AlarmCheck` subcomponent are declared. The definitions of the corresponding component types are given in in the top part of the figure. On the left side (a), the autoconnect port strategy is used. on the right side (b), the autoconnect type strategy is used. Type compatible ports of component `AlarmCheck` and `Arbiter` are ports `br` and `blink` (`BlinkRequest`), `s` respectively `string` (both of type `String`), `i1`, and `i2` (`Integer`). Strategy `port` connects the ports of both subcomponents that have the same name and compatible types. This way, the ports `i1`, and `i2` are connected. The type strategy creates connector `ac.string -> arbiter.s` but does not create connections for ports `i1` and `i2` since both source and target ports have the same type.

Expand Autoinstantiate

Inner component definitions of a component have to be declared as a subcomponent manually. The `autoinstantiate` command switches on the automatic instantiation of inner components. This transformation creates a subcomponent with the type of the inner component for

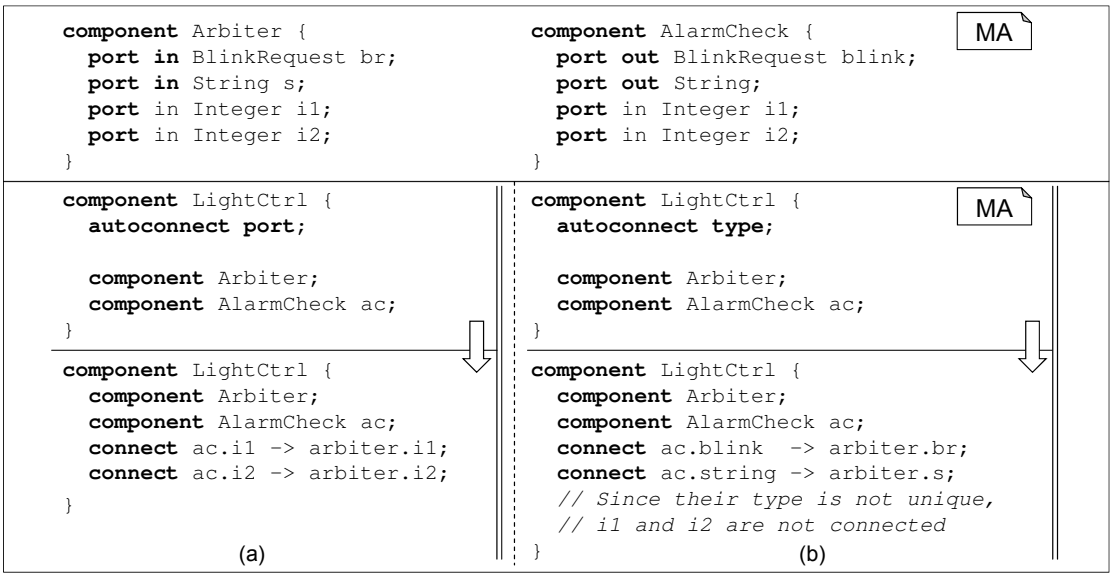


Figure 5.16: Pre CoCo Trafo: Expand autoconnect port (a) and type (b).

each inner component which is neither generic nor configurable and not manually declared as a subcomponent. Therefore, the transformation collects all used component types from all subcomponents of the current component and it collects all inner component definitions. Then, it iterates over all inner component definitions and checks whether the aforementioned precondition for automatic instantiation holds. If it holds, a new subcomponent is created and added to the component's AST and symbol table.

The transformation and its effect is demonstrated in Figure 5.17. Component `LightCtrl` contains an inner component definition `AlarmCheck`. Since `autoinstantiate` is switched on, this transformation creates a subcomponent `alarmCheck` with the type of the inner component definition. The inner component definitions `Delay` and `Buffer` are not instantiated since the former is generic and the latter configurable.

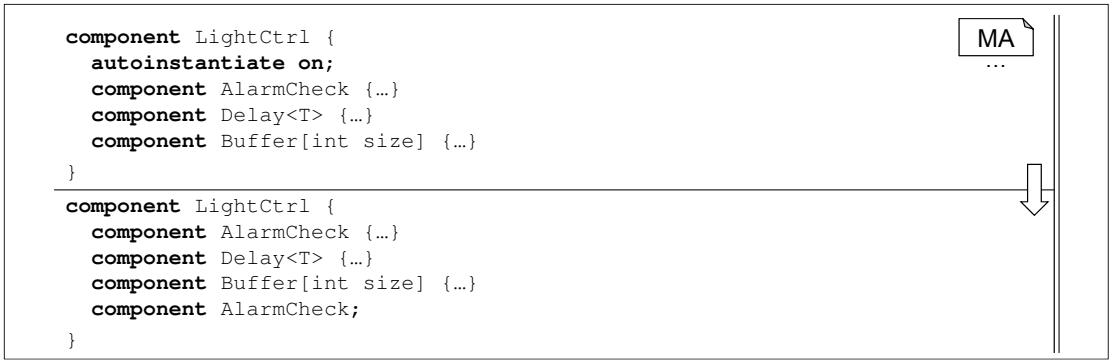


Figure 5.17: Pre CoCo Trafo: Expand Autoinstantiate.

5.3.2 Pre Code Generation Transformations

The following transformations prepare the AST for the code generation workflow. They mostly map extended concepts to basic concepts to ease code generation. In this way, the generator is simplified since it only has to handle basic concepts while ignoring extended concepts. The following transformations are executed before code generation.

Name Implicitly Named Subcomponents

This transformation transforms implicitly named subcomponents (cf. Section 3.4.3 on page 53) into explicitly named subcomponents by setting the explicit name with the derived implicit name (which is the type name starting with a small letter). This allows the code generator to only handle named subcomponents. This transformation is executed in the transformation workflow after context condition checks. Thus, inner component definitions have already been automatically instantiated by transformation **Expand autoinstantiate** (see Figure 5.17). Consequently, this transformation also transforms the subcomponents created by **Expand autoinstantiate** by assigning an explicit name to them.

The effect of this transformation is demonstrated in Figure 5.18. Component `LightCtrl` contains an implicitly named subcomponent `alarmCheck`. As a result of the transformation, the explicit name of this subcomponent is set to `alarmCheck`.

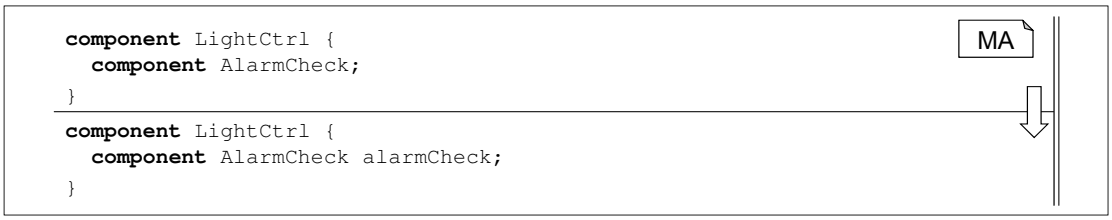


Figure 5.18: Pre Codegeneration Trafo: Name Implicitly Named Subcomponents.

Name Implicitly Named Ports

In MontiArc, implicit naming is also available for ports (cf. Section 3.4.3 on page 53). An implicitly named port is referenced by using the unqualified type name starting with a small letter. This transformation sets the explicit names of implicitly named ports.

An example for the effect of the transformation is given in Figure 5.19. Component `LightCtrl` contains an implicitly named port `alarmStatus` whose explicit name is set to `alarmStatus` by the transformation.

Expand Simple Connectors

Beside normal connectors, MontiArc also offers simple connectors which can be directly attached to a named subcomponent (cf. Section 3.4.3). Sources of a simple connector are ports from the corresponding subcomponent. To ease code generation, this transformation converts

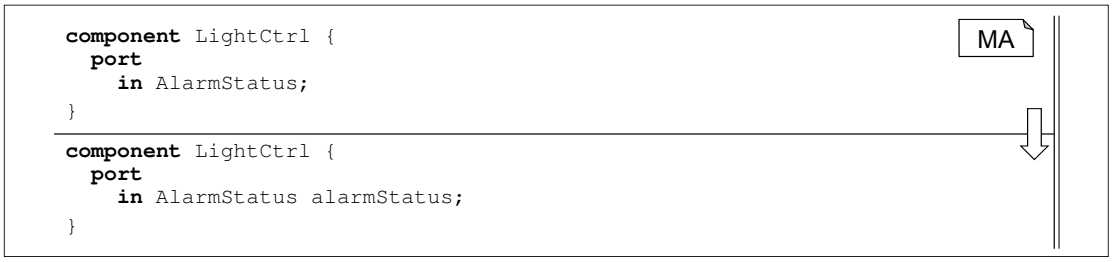


Figure 5.19: Pre Codegeneration Trafo: Name Implicitly Named Ports.

simple connectors to semantically identical normal connectors. Therefore, the simple connector is replaced with a regular connector and the source of the connector is qualified with the subcomponent's name. This way, only normal connectors have to be handled in the code generation step afterwards.

This is demonstrated in Figure 5.20. In the top part of the figure, a simple connector is attached to subcomponent `ae` that connects its outgoing port `blinkRequest` with the `blinkRequest` port of subcomponent `interiorLightEval`. The result of the transformation is depicted in the bottom part of the figure. The simple connector has been replaced with a normal connector which has the same source (port `blinkRequest` of subcomponent `ae`) and target.

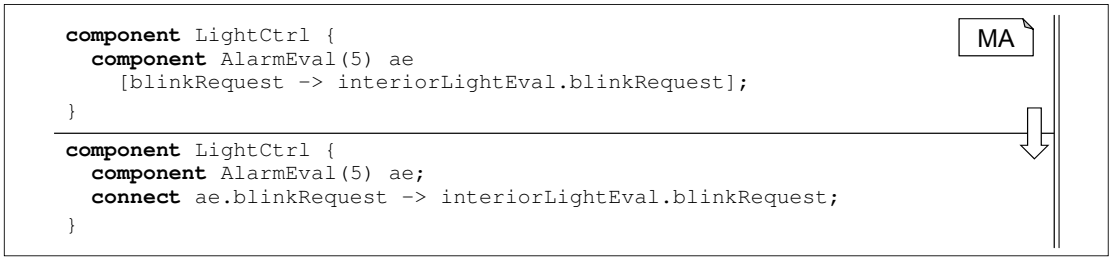


Figure 5.20: Pre Codegeneration Trafo: Expand Simple Connectors.

Qualify all Types

To avoid repeated qualification of types during code generation, all used types are qualified by this transformation directly in the AST. The following types are replaced with their qualified version:

1. configuration parameter types,
2. configuration value types (e.g., used constructors or enum values),
3. generic type arguments,
4. upper bounds of generic types,
5. port types,
6. subcomponent types, and
7. supercomponent types.

An example demonstrating an excerpt of these qualifications is given in Figure 5.21. The top part of the figure contains the definition of component `AlarmEval` which uses certain type references. It can be seen that the type of the configuration parameter `interval` is replaced with its qualified version, the type of the supercomponent `Eval` and the port types are qualified, too. The type of subcomponent `fd`, its generic type argument `BlinkRequest`, and the type of the used configuration value are qualified as well.

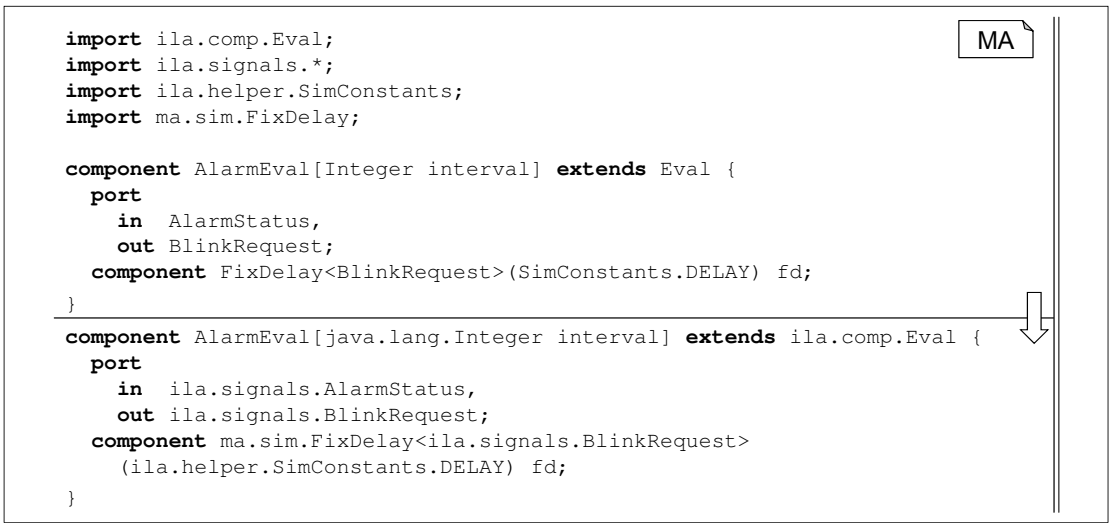


Figure 5.21: Pre Codegeneration Trafo: Qualify all Types.

Unconnected Incoming Ports of Subcomponents

Missing connections to incoming ports of subcomponents will result in a warning during context condition checks (see Section 3.5.2). However, it is not forbidden to leave ports of subcomponents unconnected. In this case, the subcomponent only provides a subset of its functionality. If all incoming signals are needed to operate properly, even no functionality is provided at all. Nevertheless, the simulation of time should still be possible. As explained in Section 4.3 on page 96, the simulation scheduler expects a $\surd$ on each incoming port of a component to emit a $\surd$ on all outgoing ports. If one incoming port of a component is not connected, it will never receive a tick which leads to a simulation deadlock.

To avoid this problem, this transformation connects unconnected ports of subcomponents with a connector that solely transmits ticks. Therefore, the decomposed component is complemented with a `TickSource` subcomponent which is triggered by the scheduler to emit ticks. Its outgoing port is connected to all unconnected ports of all subcomponents. This way, these unconnected ports are provided with ticks to avoid the aforementioned simulation deadlock.

An example for this transformation is given in Figure 5.22. The top part of the figure depicts an excerpt of component `LightCtrl` which contains subcomponent `DoorEval` that has an unconnected port. The bottom of the figure depicts the result of this transformation. It can be

seen that subcomponent `TickSource` has been inserted and connected to the unconnected port of subcomponent `DoorEval`.

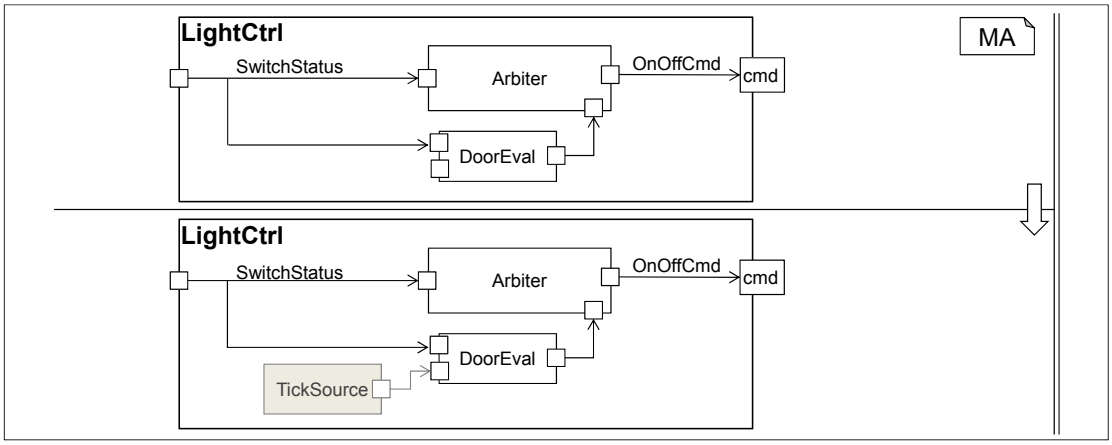


Figure 5.22: Pre Codegeneration Trafo: Connecting Unconnected Incoming Ports of Subcomponents.

Unconnected Outgoing Ports of Decomposed Components

This transformation is motivated by the same reasons like the previously described transformation. If a decomposed component has an outgoing port which is not connected to an outgoing port of a contained subcomponent, this port will never emit a message or a $\surd$. If such a component is reused as a subcomponent in the context of another decomposed component definition, it will most likely cause a simulation deadlock due to missing ticks that are not emitted by this "dead" port. To avoid this problem, this transformation will connect a `TickSource` subcomponent, which is inserted into the decomposed component, with all unconnected outgoing ports.

This transformation is exemplary demonstrated in Figure 5.23. The top part of the figure depicts component `LightCtrl` with two unconnected outgoing ports. The bottom part shows

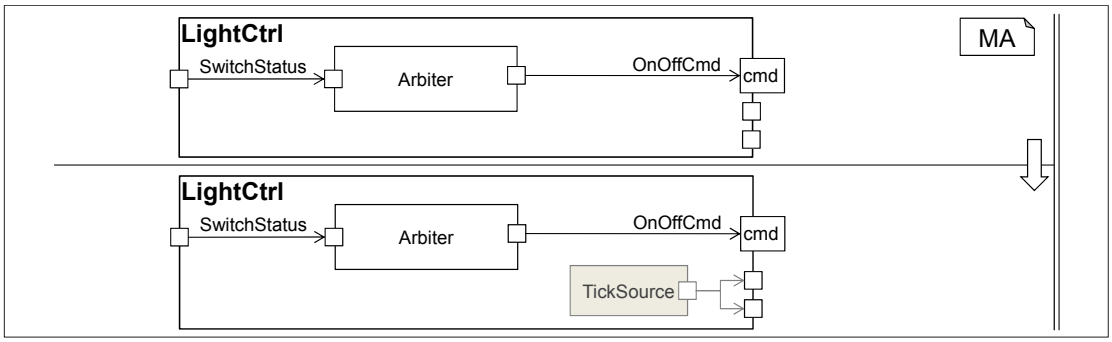


Figure 5.23: Pre Codegeneration Trafo: Connecting Unconnected Ports of Subcomponents.

the result of this transformation. It can be seen that subcomponent `TickSource` has been inserted and its outgoing port is connected to both "dead" ports. This way, these ports emit ticks provided by the `TickSource` subcomponent which is controlled by the scheduler.

Please note that only a single `TickSource` subcomponent is used to handle all unconnected outgoing ports. Since this transformation is independent from the previous transformation, the latter also introduces another single `TickSource` which handles all unconnected incoming ports of subcomponents. It would also be possible to use a dedicated `TickSource` for each unconnected port. Though, this would result in an increased scheduler effort which contradicts to requirement *SRQ10*.

5.3.3 Implementation

MontiArc provides an expandable transformation infrastructure in which further transformations can be added easily. The realized transformation framework is designed similar to the context condition check infrastructure which is presented in [Sch12, Section 7.3]. An excerpt of the structure of the transformation framework is depicted in Figure 5.24. Both previously discussed transformation phases are realized by `ConcreteTransformationWorkflows` which are responsible for executing a `TransformationVisitor` or an instance of a subclass. Using its `configure(ITrafoConfiguration)` method, the associated transformations of the given configuration are registered. For all MontiArc AST nodes, the framework contains a transformation interface and the `TransformationVisitor` contains a `visit(...)` method that visits the corresponding node. This visit method then resolves the corresponding sym-

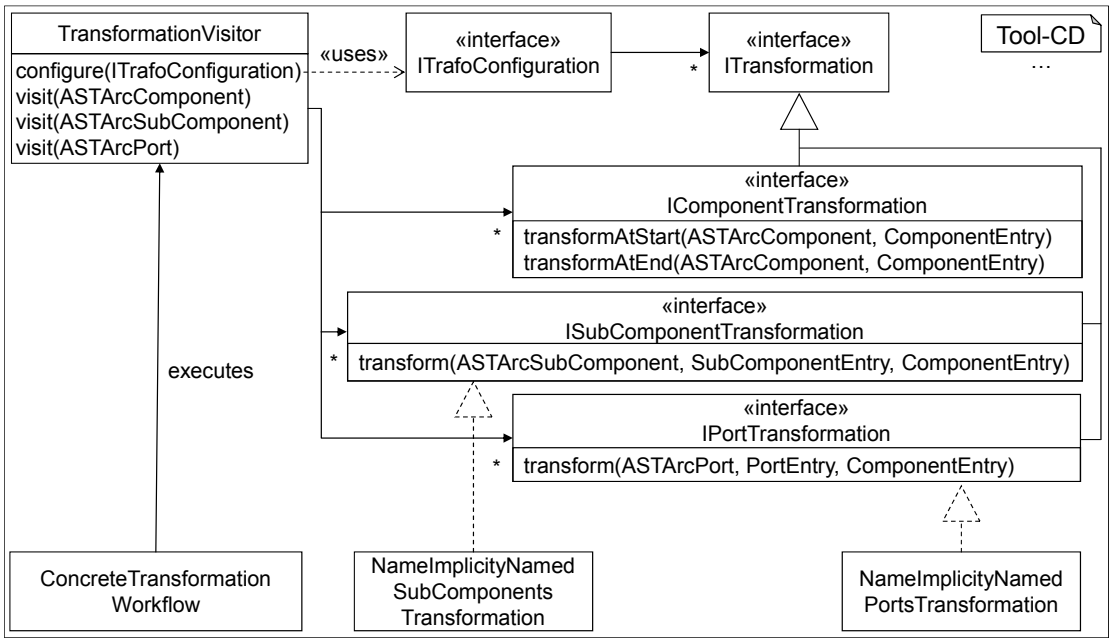


Figure 5.24: Important classes of MontiArc's transformation framework.

bol that represents this node and delegates both, node and symbol, to all registered responsible transformations. For example the `visit(ASTArcPort)` method first resolves the symbol (`PortEntry`) of the current `ASTArcPort` node and then calls the `transform` method of all registered `IPortTransformations` (e.g., `NameImplicitlyNamedPortsTransformation`). A complete list of transformation interfaces is given in Table F.4 on page 341.

All previously described transformations are realized by a corresponding concrete transformation that implements the needed transformation interfaces. An example how these transformations are realized is given in Listing 5.25 which contains the implementation of transformation "*Name Implicitly Named Ports*". It implements the interface `IPortTransformation` since it transforms ports. At first, it is checked whether the current node is implicitly named (l. 6). If this holds, the name of the node is set to the name of the corresponding port entry (ll. 7f). This is possible since the entries contained in the symbol already have the implicit name.

```
1 public class NameImplicitlyNamedPortsTransformation                    Java
2     implements IPortTransformation {
3   @Override
4   public void transform(ASTArcPort node, PortEntry entry,
5       ComponentEntry currentComp) {
6     if (node.getName() == null) {
7       String name = entry.getName();
8       node.setName(name);
9     }
10  }
11 }
```

Listing 5.25: Exemplary implementation of a MontiArc transformation.

5.4 Generation of Simulation Code

Several artifacts, which are based on the runtime environment (RTE) described in Section 4.2, are generated for each MontiArc component. By using these artifacts, the component model can be simulated. These are:

- **Component interfaces**: A Java interface which represents the port interface of the component. It handles how a component can be accessed by its environment.
- **Component factories**: A Java class responsible for creating instances of a component.
- **Component classes**: A Java class which represents the component definition in the simulation. Classes of atomic components hold port objects, classes of decomposed components hold objects of their subcomponents.

In the following, the generation of these artifacts is explained and depicted using the notation of a transformation which translates a component into Java code.

5.4.1 Component Interfaces

For each component model, a Java interface is generated that represents the port interface of the component. It contains methods to access the component's incoming and outgoing ports. Depending on the component's timing (cf. Section 4.4), a suitable super-interface is chosen.

An example for the generation of an interface of a component with instant timing is given in Figure 5.26. The component `LightCtrl` has an incoming port `switchStatus` with type `SwitchStatus` and an outgoing port `cmd` with type `OnOffCmd`. The bottom part of the figure contains the generated interface for this component, the name is prefixed with an `I`. The incoming port is compiled into a getter method which returns an `IInPort` typed with the port type `SwitchStatus`. The name of the getter method corresponds to the port's name. The outgoing port `cmd` is compiled into the getter `getCmd()` that returns a suitably typed `IOutPort` and a setter `setCmd(...)` that allows to set the component's outgoing port `cmd`. Please note that *instant* is the default timing domain of MontiArc components. Hence, the generated interface extends `ITimedComponent` (cf. Figure 4.11 on page 95). This interface is also implemented by synchronous, causal synchronous, and delayed components.

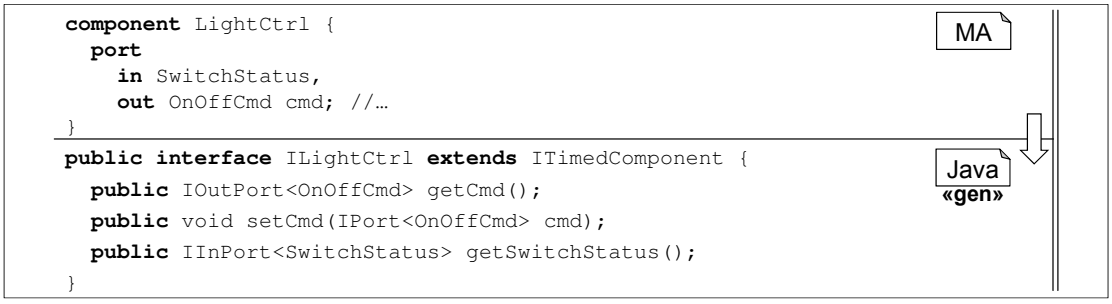


Figure 5.26: Generated interface for a timed component.

The generation of an interface of an untimed component is given in Figure 5.27. Component `LightCtrl` uses timing domain *untimed*. Hence, the generated interface extends interface `IComponent` instead of `ITimedComponent`. This way, no timing information are available for the component and its environment.

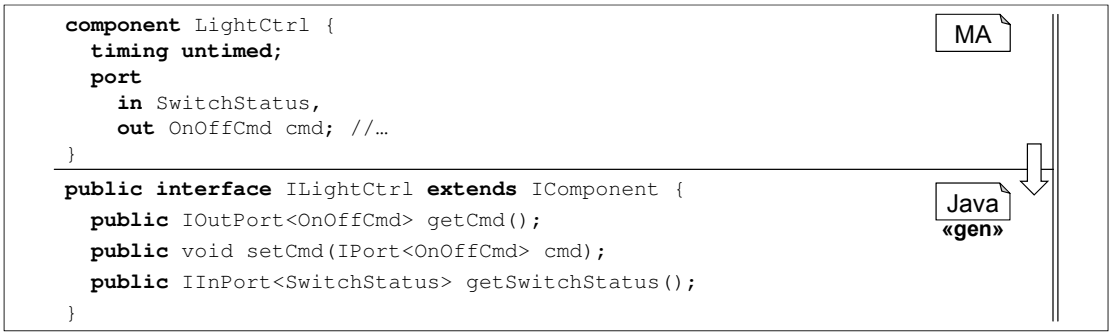


Figure 5.27: Generated interface for an untimed component.

Generic type parameters of generic components are directly mapped to Java generics. An example is given in Figure 5.28. The generic type `T` of component `Buffer` is directly represented by the generic type `T` in the Java interface. Hence, this type is also used to parametrize the getter and setter methods of the corresponding ports.

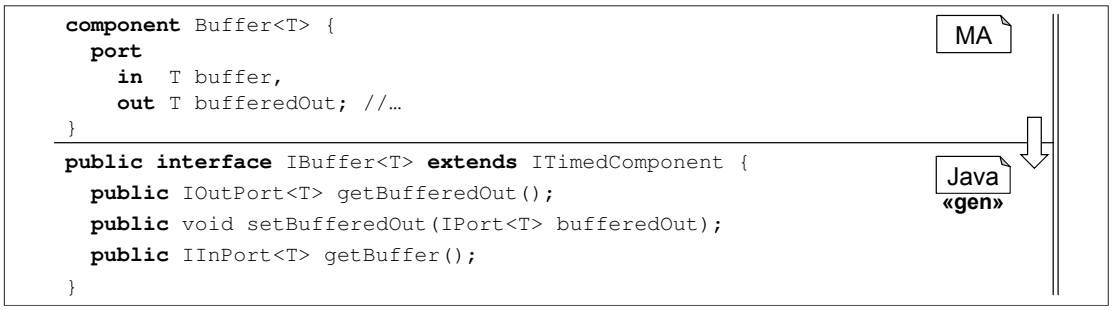


Figure 5.28: Generated interface for a generic component.

5.4.2 Atomic Components

Atomic components in MontiArc produce behavior as a reaction to events. Since the default MontiArc language does not support behavior definitions within components, behavior of atomic components has to be implemented by hand. An abstract class is generated for each atomic component which can be used as a superclass for component implementations. This class is responsible for the setup. Instances of this class hold objects of the corresponding ports. The concrete behavior has to be implemented in a subclass. The abstract class then delegates received messages and events to the concrete handwritten implementation (see Section 5.5).

Header

The header of the abstract generated class is influenced by the timing domain of the component. This is depicted in Figure 5.29. For each atomic component, an abstract class with the component's name prefixed with an `A` is generated. Therefore, for component `DoorEval` an abstract class `ADoorEval` is generated which implements the generated component interface `IDoorEval` (see Section 5.4.1). For timed components, the superclass `ATimedComponent` is used (as depicted on the left). Untimed components use the superclass `AComponent`.

Interface Implementation and Ports

The methods given in the generated component interface are already implemented in the generated abstract component class. For each incoming port, a private field with the type `IInSimPort`, for each outgoing port, a field with the type `IOutSimPort` is generated. Both are parametrized with the port's type and have the name of the corresponding port. This is demonstrated in Figure 5.30. For the incoming port `doorStatus`, a field and a getter method are generated. For the outgoing port `onOffRequest`, a field, a getter, and a setter method are

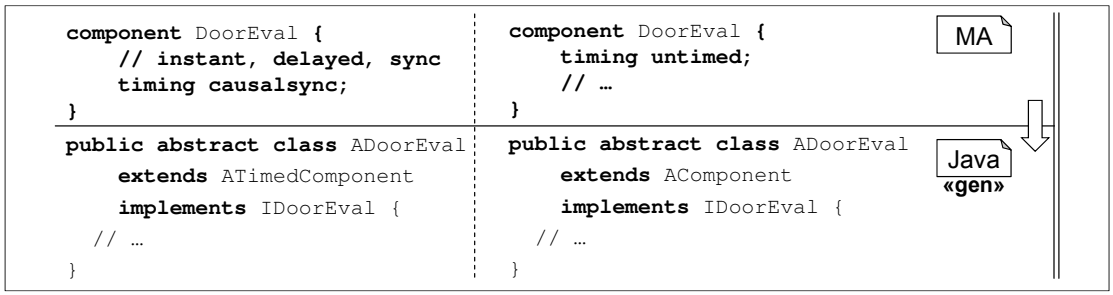


Figure 5.29: Generated abstract class for an atomic component.

generated. Since the outgoing port of a component can be connected to more than one receiver, the set method first has to check whether the outgoing port already has been set with a port object (`this.onOffRequest == null`). If it not has been set, the passed object is used as outgoing port. Else, the passed object is added to the receivers of the existing outgoing port. Please note that within a component a port is handled as a simulation port. In contrast, the getter methods return a regular incoming or outgoing port. This way, send and accept method functionality is available from outside the component. The setup and scheduling functions are only visible within the component.

Figure 5.30: Implementation of the component interface for an atomic component.

Configurable Components

Configurable MontiArc components are parametrized (cf. Section 3.2) to realize adjustable behavior depending on the passed parameters during component instantiation. This also has to be reflected in the generated code. Each defined component parameter results in a private attribute

to store this parameter, a getter method to access its value, and a parameter in the generated constructor. An example is given in Figure 5.31 that depicts configurable component `AlarmCheck` which has the configuration parameter `blinkIntervall` with type `int`. It can be seen that a private field with the same type and name has been generated, a protected getter method which returns the field. The constructor is extended with an additional parameter `blinkIntervall`. Please note, the same parts are generated for configurable decomposed components. Generated component factories of configurable components also have the component's parameters as parameters of its `create` method.

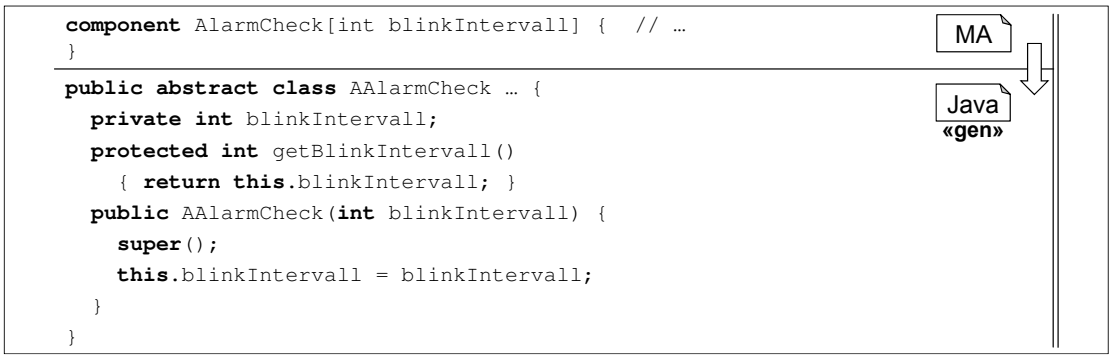


Figure 5.31: Generated code for a configurable atomic component.

Atomic Components Setup

During the setup of an atomic component, incoming port objects are instantiated and links to the scheduler and error handler are stored. This is demonstrated in Figure 5.32 for atomic component `DoorEval`. After the links to scheduler and error handler have been set, port `doorStatus` is instantiated using the factory method of the scheduler (`s.createInPort`). Finally,

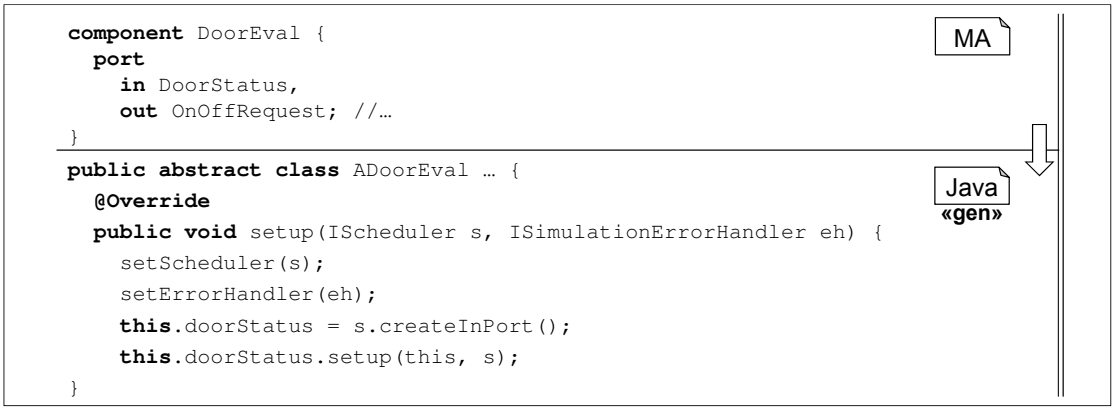


Figure 5.32: Setting up atomic components.

the `setup()` method of the port is called to initialize its relation to the component and the scheduler.

Please note, the scheduler and ports closely interact with each other. Therefore, the scheduler is responsible for port creation. If a new scheduler strategy is realized, it also has to deliver the corresponding port implementation.

Message Propagation

Section 4.4 defines the support of different timing domains with a mapping from timed streams to timing-specific component traces. This mapping is realized by the generated component code which handles the propagation of scheduled messages to the component implementation. Regarding message propagation, untimed, instant, and delayed components are handled the same way, while synchronous and causal synchronous components follow their own realization.

As explained in Section 4.3, the simulation scheduler calls the `handleMessage` method to deliver a message to a component. Message propagation to the component implementation is handled there. Hence, the aforementioned mapping is realized in this method. Figure 5.33 compares the generated code for an instant version (on the left) and a causal synchronous version of component `DoorEval` (on the right). The former checks which port accepted the current message (`p == doorStatus`) and delegates the contained data to the abstract `treatDoorStatus(...)` method which handles *data events* on this port. This method has to be realized in

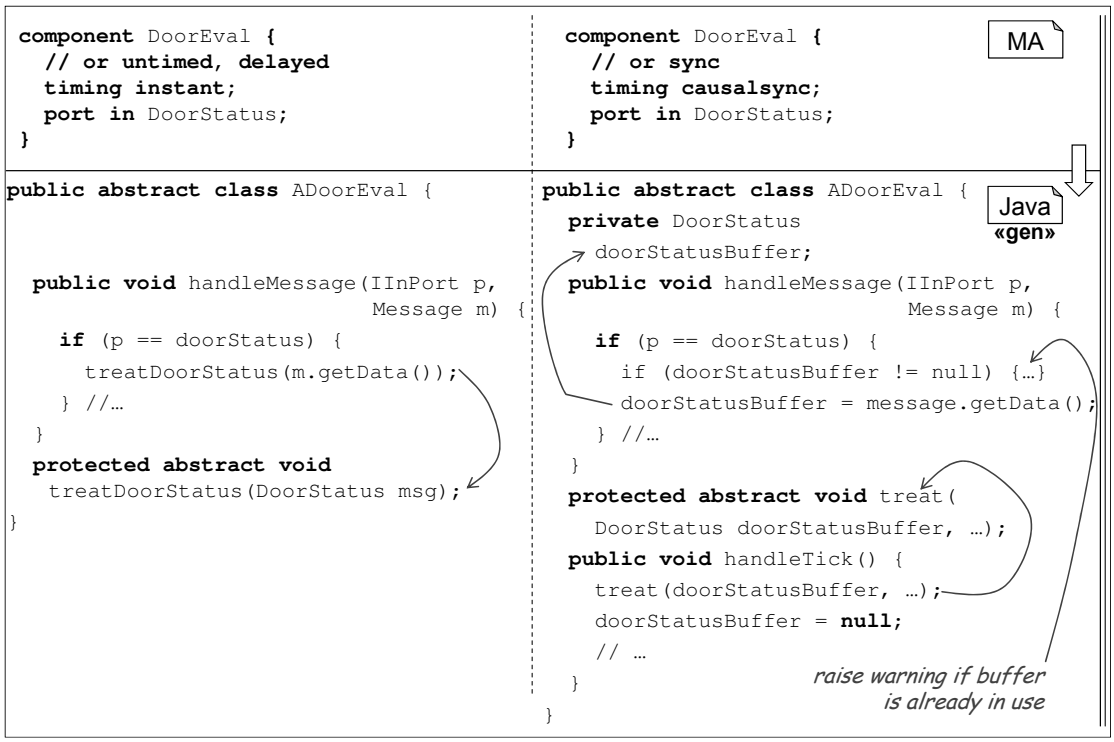


Figure 5.33: Message propagation of atomic components.

an handwritten subclass.

In contrast, synchronous components exclusively handle *data event tuples*. These events compose a single data object for each port to an event tuple. If more than one message is accepted on a single port, the last message is used for this composition (cf. Section 4.4.4) and a warning is raised. Hence, a message buffer is generated for each incoming port (e.g., `doorStatusBuffer`) which is used to store messages passed by the scheduler to the `handleMessage` method. If the corresponding buffer is already in use, a warning is raised. The data event tuple is created by delegating the content of all incoming port buffers to the abstract `treat(...)` method that has a parameter for each incoming port. This method is to be implemented manually in a subclass and is automatically called within the `handleTick()` method which synchronizes the event creation. Afterwards, all buffers are erased for the next time interval by assigning a `null` value. Thus, $\perp$ events in time-synchronous streams, which are used to model that no message is transmitted within a time interval, are represented by `null` values of erased buffers.

Message Sending

Again, sending of messages as a reaction to incoming data and time events is uniformly implemented for untimed, instant, and delayed components. The sending of messages of synchronous and causal synchronous components is realized slightly different. Figure 5.34 compares both to each other. For the former, a `send` method is generated for each outgoing port of the component. It encapsulates the message which is to be sent into a `Message` object using the `Message.of()` method. The encapsulated message is then delegated to the `send()` method of

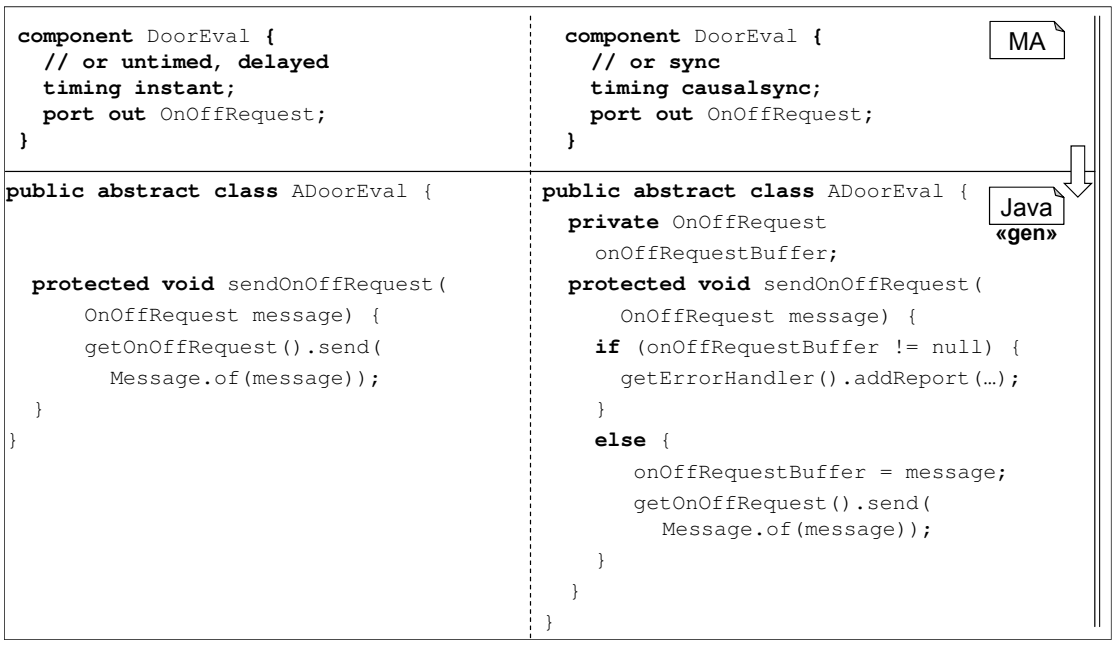


Figure 5.34: Message sending of atomic components.

the corresponding port. This is depicted on the left side of the figure. For port `onOffRequest` a method `sendOnOffRequest()` is generated which takes an `OnOffRequest` object as parameter. If this method is called by the handwritten behavior implementation, the passed object is encapsulated into a `Message` object. The message is delegated to the `send()` method of port `onOffRequest` which is accessed by the corresponding get method.

Synchronous components are allowed to emit at max one message on each outgoing port in each time interval. To prevent handwritten implementations to send more than one message, sent messages are buffered in a single message buffer. Such a buffer is generated for all outgoing ports of a synchronous component. The buffer of a port is filled in the generated `send()` method of the port before the message is emitted. If the buffer is already filled, and thus a message has already been emitted on the same port, a warning is raised. At the end of a time interval this buffer is erased.

Tick Handling

If a component consumes a tick on all incoming ports, it has to emit a tick on all outgoing ports (cf. Section 4.4). As discussed in Section 4.3.2, emitting ticks is triggered by the simulation scheduler by calling the `handleTick()` method of a component. An example of such a generated method is depicted in Figure 5.35. The method obtains a tick by calling the static method `Tick.get()` and sends it via the corresponding outgoing port. Afterwards, the local clock is incremented (`incLocalTime()`) and a time event is propagated to the component by calling the `timeStep()` method that implies the start of a new time interval. Please note, `Tick.get()` always returns the same centrally managed tick object (see Section 5.6.3).

Figure 5.35: Generated method handleTick() that emits ticks at the end of a time interval.

5.4.3 Decomposed Components

Decomposed MontiArc components do not produce behavior themselves. The behavior which is observed on their interfaces is a result of the composed behavior of the contained subcomponents. Therefore, MontiArc generates a concrete Java class for each decomposed component that

is responsible for the instantiation and setup of the contained subcomponents and the creation of connections between the subcomponents and the outer ports.

Ports

Decomposed components actually do not need own port objects. They simply reuse the ports of inner subcomponents for their own interface. This, however, is only possible for outgoing ports that are connected to an unambiguous outgoing port of a subcomponent (see Section 3.5). Incoming ports of a decomposed component can be connected to more than one receiver. For this reason, an own forwarding port object as an instance of `IForwardPort` is created for each incoming port of a decomposed component (see Section 4.2.2).

An example that demonstrates which code parts are generated for incoming ports of decomposed components is given in Figure 5.36. It can be seen that no code is generated for the components outgoing port `cmd`. For the incoming port `switchStatus`, a private forwarding port field is generated which is returned in the corresponding port getter method. During the setup of the component, this field is instantiated with a port object that is created by the scheduler. Afterwards, the port is set up.

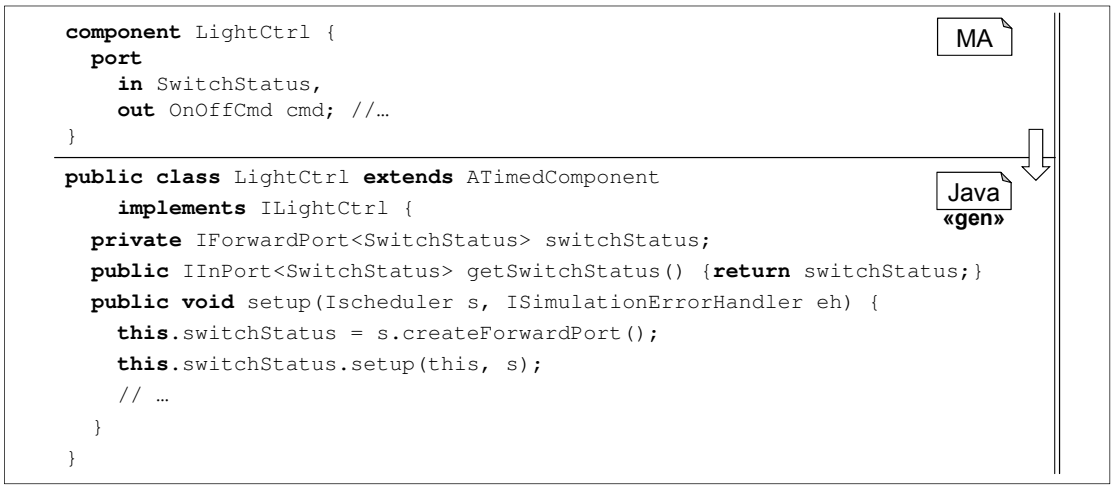


Figure 5.36: Incoming ports of decomposed components.

Subcomponents

A decomposed component holds a private field for each contained subcomponent. These fields have the interface as type that has been generated for the corresponding subcomponents component type (see Section 5.4.1). The field that represents a subcomponent is instantiated using the corresponding generated factory (see Section 5.5.2) and set up during the setup of the decomposed component. If the subcomponent is parametrized with configuration parameters, these parameters are also passed to the component factory during instantiation. Consequently, component instances are created from top to bottom of the component hierarchy as described in

Section 4.2.1.

This is demonstrated with the aid of a simple example given in Figure 5.37. It is depicted that a private field and a protected getter method is generated for both subcomponents `arbiter` and `ac`. In the excerpt of the `setup` method, it is shown how instances are created using the corresponding component type factories. The configuration parameter `5`, which is used for subcomponent `ac`, is passed to the `AlarmCheckFacotry` to configure the produced instance. After a subcomponent instance has been created, its `setup` method is called to set up the subcomponent with the used scheduler `s` and error handler `eh`.

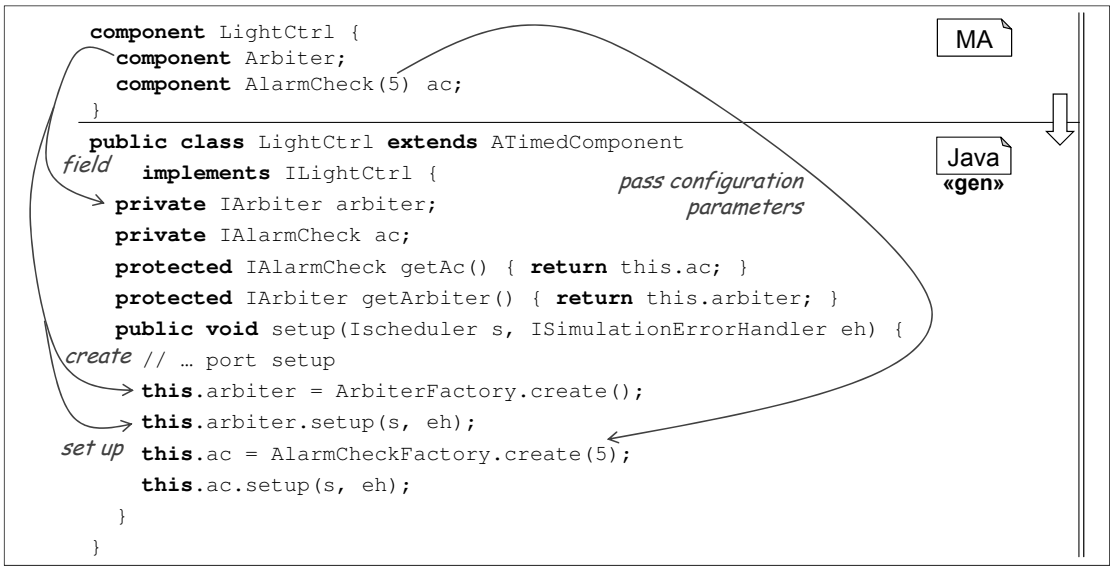


Figure 5.37: Code generation for subcomponents.

Connectors

Three different kinds of connectors exist in MontiArc. The first kind connects an incoming port of a decomposed component with an incoming port of a contained subcomponent. The second kind connects two subcomponents of a decomposed component. And the third kind connects an outgoing port of a subcomponent with an outgoing port of a decomposed component. An example for all three connection kinds is depicted in Figure 4.3 on page 90.

As explained in Section 5.4.3, incoming ports of decomposed components are represented by `ForwardPorts` in the generated code. The first kind of connectors is realized by adding the incoming port of the subcomponent to the receivers of the forwarding port of the decomposed component. This is demonstrated in Figure 5.38. The shown connector connects port `switchStatus` with port `switchStatus` of subcomponent `arbiter`. Therefore, in the setup method generated for component `LightCtrl`, arbiter's port `switchStatus` is added to the receivers of the forward port `switchStatus` using its `add(...)` method. This is done after the ports of the current component and all subcomponents are set up as described in the previous sections.

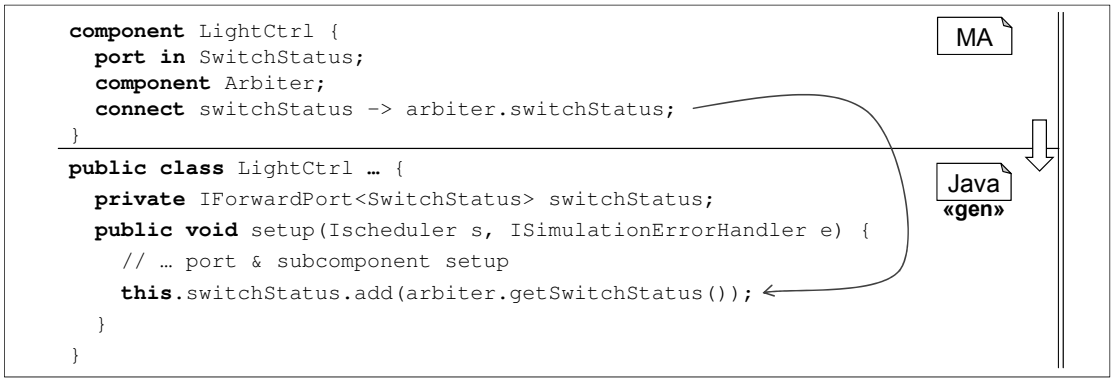


Figure 5.38: Generated code for connectors from incoming ports of decomposed components to incoming ports of subcomponents.

The second kind of connections is realized by using the incoming port of the receiver as outgoing port of the sending subcomponent. This is possible since ports respectively port objects always serve as incoming and outgoing ports by implementing both corresponding interfaces `IOutPort` and `IInPort` (see Figure 4.10 on page 94). Hence, the same port object is shared between both subcomponents used as an outgoing port by the sender and as an incoming port by the receiver. If more than one port is connected with the same outgoing port, additional incoming ports are registered as further receivers in the `receivers` set of the outgoing port (see Figure 5.30). An example for the generated code for this connector kind is given in Figure 5.39. In the setup method of component `LightCtrl`, the port `onOffRequest`, which belongs to the sending subcomponent `doorEval`, is set to the incoming port `onOffRequest` of subcomponent `arbiter` using its `setOnOffRequest(...)` method. Please note that the casting of the incoming port to `IPort` is needed since set methods, generated for outgoing ports, expect an `IPort` argument, but get methods, generated for incoming ports, return an `IInPort` object (see Section 5.4.1). This cast is typesafe since all port objects in the simulation are an instance of `IPort` (see Figure 4.10).

Figure 5.39: Generated code for connectors which connect two subcomponents.

The third connector kind, which connects outgoing ports of subcomponents with outgoing

ports of decomposed components, is realized the following way. As well as atomic components, decomposed components do not have own port objects for outgoing ports. Therefore, the generated getter and setter methods for outgoing ports of decomposed components simply delegate to the connected port of the corresponding subcomponent. An example is given in Figure 5.40. The methods `getCmd()` and `setCmd(...)`, generated for the outgoing port `cmd`, delegate to the get (`getOnOffCmd()`) respectively set (`setOnOffCmd()`) method of the sending port which belongs to subcomponent `arbiter`.

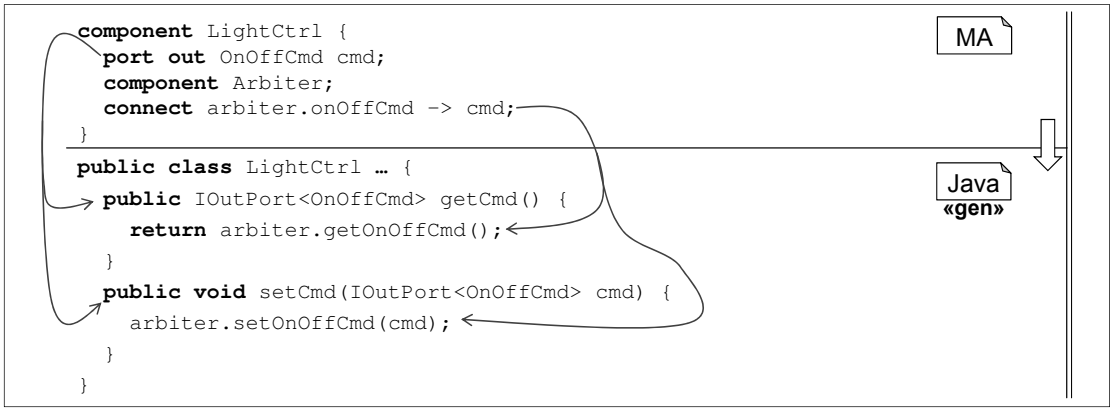


Figure 5.40: Generated code for connectors which connect outgoing ports of subcomponents with outgoing ports of decomposed components.

5.5 Atomic Component Behavior Implementation

Decomposed MontiArc components do not implement behavior themselves. Their shown interface behavior is the result of the composition of the behavior generated by the contained subcomponents. Atomic components, that are not decomposed anymore, have to implement behavior themself. Since MontiArc does not include a language that allows the implementation of behavior within components, the behavior has to be implemented externally in Java, the target language of the MontiArc simulation.

5.5.1 Implementation

To ease the implementation of atomic component behavior, MontiArc generates abstract superclasses which already handle message transmission and event forwarding from the simulation scheduler (see Section 4.3). This way, the component developer does not have to care about message transmission, he only has to handle events and data messages. Depending on the component's timing domain, the abstract generated superclass provides event methods which have to be implemented in the concrete behavior implementation. It is distinguished between instant, delayed, and untimed timing implementations on the one hand and (causal) synchronous timing implementations on the other hand. The former handle *data events*. *Time events* are additionally

handled by instant and delayed components while being ignored by untimed components (see Section 4.4). Synchronous components handle *data event tuples*.

Data Event Implementation

Data events are data messages which are received on a single incoming port of a component (see Definition 4.3 on page 106). They are processed by instant, delayed, and untimed components. The abstract generated superclass for these timing domains contains an abstract treat*PortName* method for each incoming port of the component. These methods, that handle data events raised at the corresponding port, have to be implemented in a concrete behavior implementation. To emit messages, a send*PortName* method is provided that immediately sends the given message via the corresponding port.

Figure 5.41 depicts the relevant artifacts needed to implement the behavior of the untimed component `DoorEvalUntimed`. This component evaluates the state of the doors if the switch status is `DOOR_DEPENDENT`. It can be seen that the generated superclass contains the abstract method `treatDoorStatus` for the incoming port `doorStatus` and a corresponding abstract treat method for port `switchStatus`. To send messages via the outgoing port `onOffRequest`, the concrete method `sendOnOffRequest()` is provided.

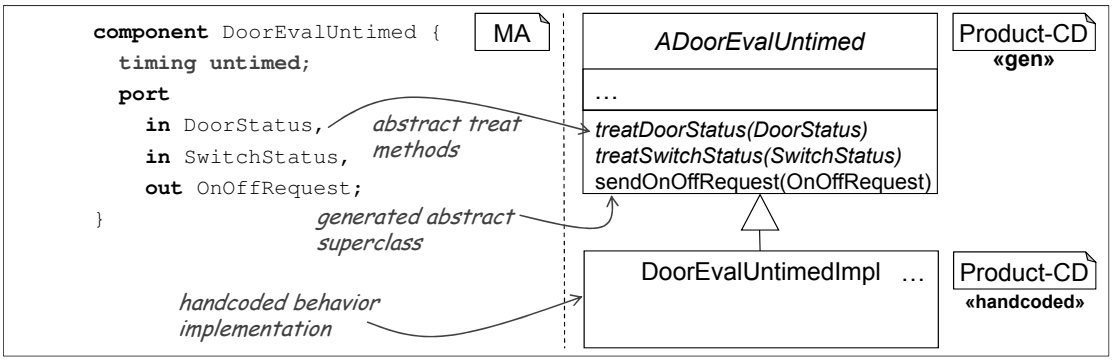


Figure 5.41: Atomic component `DoorEvalUntimed`, the corresponding generated superclass, and the implementation.

One possible behavior implementation of component `DoorEvalUntimed` is depicted in Listing 5.42. The shown class contains two private fields `currentSwitchState` and `processedDoorState` (ll. 2f) that represent the state of the component. The former is used to store the current state of the switch in method `treatSwitchStatus` (ll. 22-25). The latter is used to store the last processed door state to react to door state changes exclusively (ll. 6f). In case the switch is set to `DOOR_DEPENDENT` (l. 8), it reacts to the current change by sending an `OFF` request if the door is closed (ll. 10f). It further sends an `ON` request if the door is opened (ll. 13f). In any other case the component simply does nothing.

Java «handcoded»

```
1 public class DoorEvalUntimedImpl extends ADoorEvalUntimed {
2   private SwitchStatus currentSwitchState;
3   private DoorStatus processedDoorState;
4   @Override
5   protected void treatDoorStatus(DoorStatus message) {
6     if (message != processedDoorState) {
7       processedDoorState = message;
8       if (currentSwitchState == DOOR_DEPENDENT) {
9         switch (message) {
10        case CLOSED:
11          sendOnOffRequest(OFF);
12          break;
13        case OPENED:
14          sendOnOffRequest(ON);
15          break;
16        default:
17          break;
18        }
19      }
20    }
21  }
22  @Override
23  protected void treatSwitchStatus(SwitchStatus message) {
24    currentSwitchState = message;
25  }
26 }
```

Listing 5.42: Implementation of atomic component `DoorEvalUntimed` which handles single data events.

Data Event Tuple Implementation

Data events tuples are created by aggregating messages from all incoming ports of a component (see Definition 4.5 on page 109). Such events are restricted to contain at max one message for each incoming port of the event processing component. If no message has been received on a port, a *null* value represents the absence of a message. Data event tuples are processed by synchronous and causal synchronous components. The abstract generated superclass for these timing domains contains an abstract `timeStep` method which has a data parameter for each incoming port of such a component. This method is automatically called by the scheduler with the received data messages. If no message has been transmitted on a certain port, *null* is passed as the corresponding argument. To emit messages, a send*PortName* method is provided for each outgoing port which sends the given message via the corresponding port. If this method is called multiple times during a time interval, only the message from the first call will be transmitted and a warning is raised.

Figure 5.43 depicts the relevant artifacts needed to implement the behavior of the synchronous component `Adder`. It synchronously processes integer values received on its incoming ports

addend1 and addend2, computes the sum, and emits it via port sum. The abstract method treat has a parameter for each incoming port and handles the data event tuples. It has to be implemented in the handwritten subclass. The method sendSum is used to emit messages via port sum.

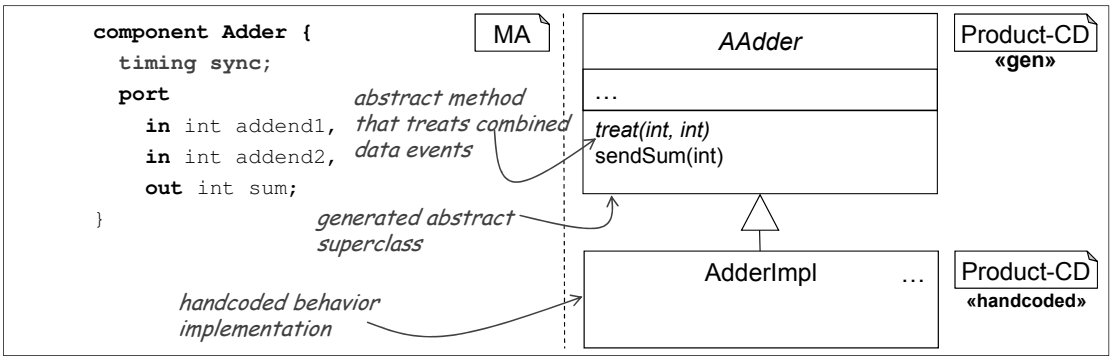


Figure 5.43: Synchronous atomic component Adder, the corresponding generated superclass, and the handwritten implementation.

An exemplary implementation of the causal synchronous Adder component is given in Listing 5.44. If a certain port has not received a value in the current time interval, the missing value (null) is mapped to zero (ll. 5f). Then, the sum is emitted via port sum using method sendSum.

Java «handcoded»

```
1 public class AdderImpl extends AAdder {
2   @Override
3   protected void timeStep(Integer add1, Integer add2) {
4     // map missing values to zero
5     int a1 = (add1 != null ? add1 : 0);
6     int a2 = (add2 != null ? add2 : 0);
7     sendSum(a1 + a2);
8   }
9 }
```

Listing 5.44: Implementation of synchronous atomic component Adder.

Time Event Implementation

Explicit time events (see Definition 4.4 on page 106) are processed by instant or delayed components and are propagated to the component if it has received a $\surd$ on each incoming port. As depicted in Figure 5.45, such components have to implement the abstract method timeStep which is located at the RTE superclass ATimedComponent. This method is automatically called at the start of a new time interval.

The figure also depicts the instant component Timer which can be used as a timed trigger. The timer is set via the port setTimer, counts down the set amount of time intervals, and emits

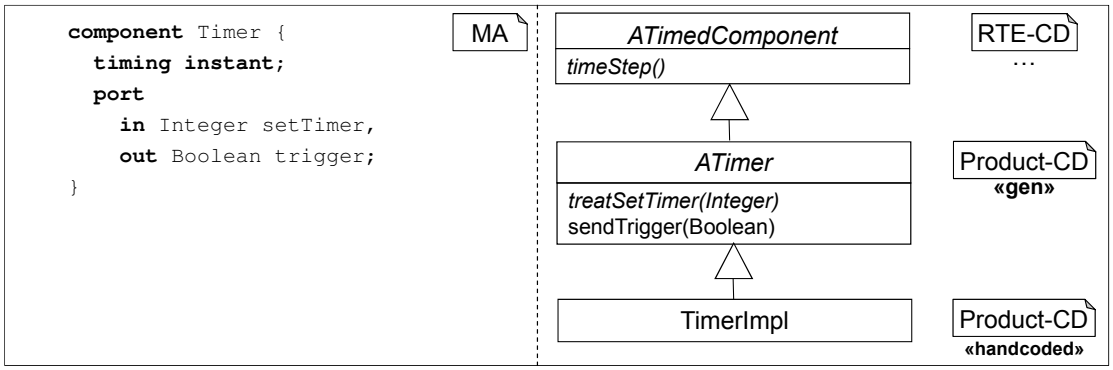


Figure 5.45: Timed component `Timer`, the corresponding generated superclass, and the handwritten implementation.

a trigger signal via its outgoing port `trigger`. Since it is an instant component, the handwritten behavior implementation has to implement the method `treatSetTimer`, to handle single data events, and the method `timeStep`, to handle time events.

An exemplary behavior implementation is depicted in Listing 5.46. A private field `countDown` represents the state of the component (l. 2). It is initialized with `-1` to start the component in an inactive state. If a `setTimer` event is received, the timer is set to the newly received value

Java «handcoded»

```
1 public class TimerImpl extends ma.util.gen.ATimer {
2   private int countDown = -1;
3   @Override
4   protected void treatSetTimer(Integer message) {
5     countDown = message;
6     if (countDown < 0) {
7       // Send timer deactivated.
8       sendTrigger(false);
9     }
10  }
11  @Override
12  protected void timeStep() {
13    if (countDown > -1) {
14      countDown--;
15    }
16    if (countDown == 0) {
17      sendTrigger(true);
18    }
19  }
20 }
```

Listing 5.46: Implementation of atomic timed component `Timer` which handles single data and time events.

(ll. 4f). If the received message is negative, a `false` is emitted to confirm the deactivation of the timer (ll. 6-9). If a time event occurs and the value of the timer is greater than `-1` (ll. 12-15), the timer is decremented. If the state variable is equal to zero after the decrement, the timer emits a `true` via its port trigger to indicate that the configured time has passed (ll. 16-18).

5.5.2 Integration of Handwritten Code

The handwritten implementations of atomic components have to be used in the simulation to produce the desired component behavior. However, the MontiArc code generator and the generated simulation classes do not know in which classes these behavior implementations are located. One possible solution is to prescribe the name of the implementation. This way, the implementation name can be uniquely derived from the qualified component name and the corresponding constructor can be directly called within the generated decomposed components. This solution is convenient because the simulation user does not have to configure the simulation in any way. But if, e.g., for testing or stubbing, more than one implementation shall exist for a single component, this solution is not feasible. Especially due to the hard coupling caused by the direct constructor calls, the component implementation can not be exchanged. A valuable solution has to realize low coupling between generated component classes and handcoded implementations. Further, it has to be convenient by minimizing needed configuration. The realized generation gap (see [GHK+15]) enables low coupling between the generated component interface and the handwritten implementation. Since instantiation of component objects is performed by dedicated factories, the needed configuration effort is minimized.

Component Factories

To decouple decomposed components from the implementation of their contained subcomponents, the abstract factory pattern combined with a singleton is suitable [GHJV95]. Beside the interface, a factory is generated for each component which is responsible for creating instances of the simulation component. This factory is generated for atomic as well as decomposed components. In a decomposed component, exclusively the generated interfaces and factories of the contained subcomponents are known and used. This way a decomposed component is decoupled from the implementations of the subcomponents.

The structure of the generated factory for component `DoorEval` is depicted in Figure 5.47. The factory contains a private static singleton field `theInstance`, which is by default initialized with an instance of the generated factory. The static `create` method, that is later on used in decomposed components to create its subcomponent instances, delegates all calls to the protected dynamic factory method `doCreate` of the object `theInstance`. This dynamic method is responsible for the concrete object creation. The static register method can be used to register handwritten factories which subtype the generated factory. It sets `theInstance` to the passed factory object. This way, calls of the static `create` method are then delegated to the `doCreate` method of the registered factory and not to the generated default factory. The `reset` method resets the singleton instance to the default factory. This way, the default factory can be used again after another factory has been registered.

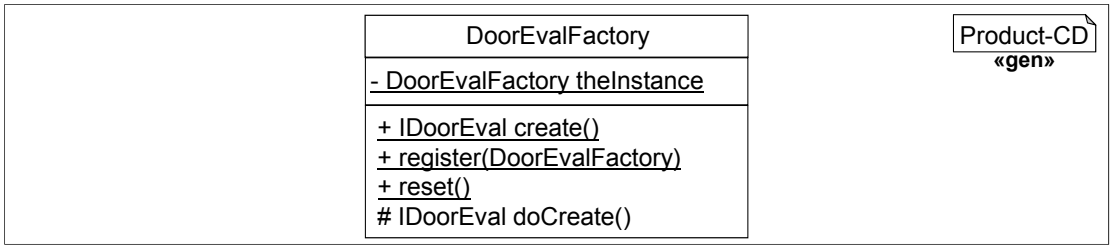


Figure 5.47: Generated factory for component `DoorEval`.

Naming Conventions

To ease the usage of the generated simulation code and to not force the user to implement a handwritten factory for each atomic component, the generated default factory is already capable to produce component objects with its `doCreate` method. Following the common design paradigm "convention over configuration", which is used by frameworks, such as Ruby on Rails[www14q] or Maven[www14k], to minimize configuration overhead, a naming convention is introduced which derives a default behavior implementation name from a component name. This default name corresponds to the qualified component name with the postfix *Impl*. Following this naming convention, the default behavior implementation name for component `DoorEval` is `DoorEvalImpl`.

Sinve the generated code of a factory shall be compilable in any case and reflections should be avoided for object creation, the code generator has to handle the following three situations while generating the `doCreate` method of the default factory:

1. The current component is decomposed.
2. The current component is atomic and the default implementation exists.
3. The current component is atomic and the default implementation does not exist.

The first case is simply handled by generating a constructor call to the generated concrete implementation of the decomposed component. The second case is handled similar, but a constructor call to the default implementation is generated. The last case is handled by generating a throw of a runtime exception which informs the user to either create a default implementation or to register a handwritten factory. This way, the generated code is compilable in any case with the drawback that factories have to be regenerated, after a new behavior implementation has been added to the project.

Custom Implementation Names

To inject an implementation into the simulation which does not obey to the naming conventions, a handwritten factory which instantiates such an implementation has to be realized. This factory has to be a subclass of the default factory and has to be registered at the default factory using its `register` method.

Figure 5.48 exemplary depicts how to inject implementations with custom names. The handwritten class `CustomDoorEvalFactory` extends the generated default factory `DoorEvalFactory`. It provides a static `init` method and overwrites the dynamic `doCreate` method.

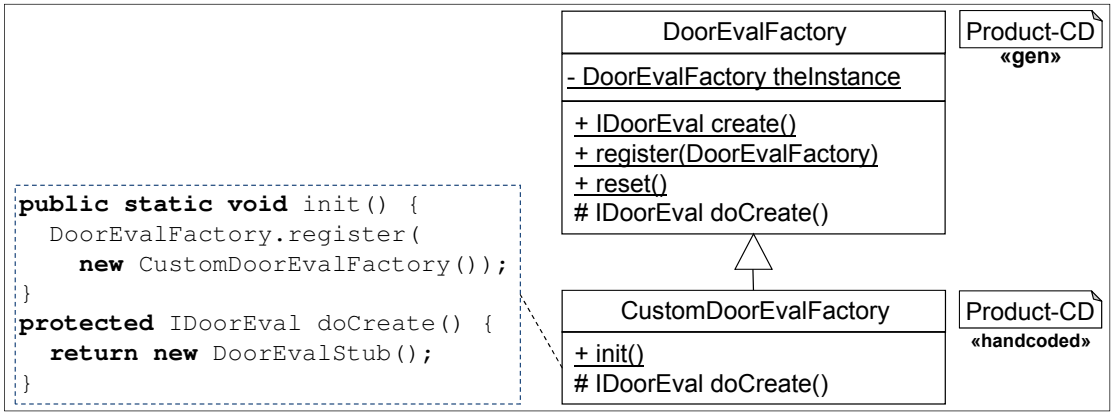


Figure 5.48: Custom factory to inject behavior implementations with custom names.

The implementation of these methods is depicted in the dashed box on the left side. The `init` method registers a new instance of the custom factory at the default generated factory and the `doCreate` method returns a new instance of the `DoorEval` behavior implementation `DoorEvalStub`. This way, the default generated factory `DoorEvalFactory`, which is used by all decomposed components that contain a subcomponent with type `DoorEval`, will produce `DoorEvalStub` instances instead of `DoorEvalImpl` instances.

5.5.3 Components with Side-Effects

As generally stated by requirement *LRQ1.1.5*, components encapsulate their state and solely communicate via their defined interface. However, the implementation of atomic component behavior in Java offers many possibilities to externalize the state of a component and to bypass the event-based communication paradigm.

Definition 5.1 ***Dirty component.*** *A dirty component has an implementation with side effects. It propagates the state of the component to its environment without using the component interface or accesses state from other components or the underlying operating system. This comprises the following techniques:*

- *Read/Write access to non-final static variables.*
- *Sharing stateful objects.*
- *Manipulation of the file system.*
- *Read/Write access to external services.*
- *Interaction with the operating system.*

All these techniques have in common, that they can be misused to directly communicate between component instances without using ports. While a controlled use of dirty implementations can be useful for architectural programing, careless use of these concepts has serious impact on the components and the FOCUS simulation. Consider the example given in Figure 5.49. Component `A` has two subcomponents that instantiate the *instant* component types `B` and `C`. The

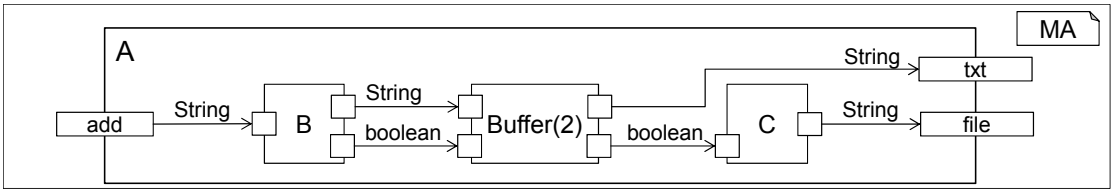


Figure 5.49: Exemplary dirty components.

Time Interval	Value add	Content Foo.txt	Value txt	Value file
0	a	a		
1	b	ab		
2	c	abc	a	abc
3	d	abcd	ab	abcd

Table 5.50: Impact of dirty implementations on the simulation time.

subcomponents are interconnected with a `Buffer` of size two. It buffers messages until the buffer size is reached, then it emits messages in a FiFo manner. `B` accepts a String, appends it to file `Foo.txt`, and sends a trigger to denote that the file has been updated (`boolean`). Further, the content of the file at the current time interval is emitted (`String`). After passing the buffer, this content is emitted via port `txt` of component `A`. When `C` is triggered, it reads the content of `Foo.txt` and emits the content of the file. Thus, the content of the file and the trigger, both emitted by `B`, arrive at the same time interval.

While this example is obviously constructed, it demonstrates the problem of side effects within MontiArc simulations. Table 5.50 depicts the values received at port `add`, the content of file `Foo.txt`, as well as the emitted values of the ports `txt` and `file` in the course of simulation time. It can be seen that the emitted values of the ports `txt` and `file` differ. While the former considers the modeled processing delay and emits the expected values, the latter bypasses the simulation time and thus ignores the modeled delay. Similar effects can be achieved with the other techniques listed above.

Beside the impact on the simulation time, another negative effect can be observed. `C` depends on the existence of file `Foo.txt`. Thus, a hardwired coupling between `C`, which consumes the file, and `B`, which produces the file, is introduced. Consequently, a `C` instance is only functional together with an instance of `B`, which contradicts to requirements *LRQ1.2* and *LRQ1.3* since `C`'s component interface does not reflect this dependency. If multiple instances of `B` and `C` are active within a single system, unforeseen and non-replicable effects can occur.

Nonetheless, dirty components still have their use in architectural programming. For example, generated reports can be sent via an e-mail message, websites or the file system can be observed to trigger a process, or a data base can be accessed. The following advice shall be considered when developing dirty components:

1. At first, answer the question, whether the introduced side effect is absolutely necessary. If

the same function can be realized without side effects, choose the implementation without side effects.
2. Use inner component definitions to encapsulate dependable components, e.g., `B` and `C`, into the same containing component. In this way, dependable components are always instantiated together with the containing component.
3. Always consider that a dirty component can be instantiated multiple times. Do multiple instances influence each other with their side effects?
4. To avoid conflicts between multiple instances of dirty components, introduce configuration parameters which allow to configure the shared resource. For example, a configuration parameter `File resource` can be added to the components `A`, `B`, and `C`. `A` then forwards this parameter to the instances of the subcomponents. Consequently, for each instance of `A`, a distinct shared resource can be assigned.
5. To support the reuse of a dirty component, precisely document the side effects. For this purpose, MontiArc's documentation generator provides the tag `@sideEffects`. Further information about component documentation are given in Section 6.7.2.

To not accidentally introduce side effects, the following advice shall be generally considered for component implementations:

1. Do not read from and do not write to static non-final fields.
2. The state of a component, and thus the reaction to incoming data events, shall only depend on primitive fields, private unshared objects, or immutable shared objects. Please note, components can technically share objects by sending them via a connector to another component since only the pointer to the object is transmitted (see Section 4.6). Thus, if the state of a component is influenced by an object, it shall not be transmitted to another component or it shall be immutable.
3. Do not communicate over the file system.
4. Be careful with object instances which are created by factories or are injected. Depending on the corresponding factory implementation or the context, these objects can be singletons [GHJV95] which are shared among all component instances of a system.

5.6 Reduction of Redundant Objects

As claimed by requirement *SRQ10*, memory inefficiencies are to be avoided. Thus, optimizations are introduced for certain component patterns and scenarios to reduce the amount of instantiated simulation objects. The realized optimizations, which are performed by the code generator, are discussed in the following.

5.6.1 Atomic Components with a Single Incoming Port

The target of this optimization is to reduce the amount of needed port objects in atomic components. Since the incoming port object from a connected component is used as outgoing port, no dedicated objects for outgoing ports are needed. Thus, only the amount of incoming port objects can be reduced. This is realized by using specialized components provided by the RTE which

also serve as incoming ports. This is possible if an atomic component has only one incoming port. In the following these components are called *single-in components*.

The extensions of the RTE to provide specialized superclasses for single-in components are depicted in Figure 5.51. The abstract class `ASingleIn` combines the properties of a port by extending class `Port` and the properties of a component by implementing interface `ISimComponent` (a description of these RTE elements is given in Section 4.2.2). This abstract class is used as superclass for the generated component class instead of the abstract class `AComponent` if the code generator detects an untimed single-in component. For timed single-in components, the RTE provides superclass `ATimedSingleIn`. It extends class `ASingleIn` and additionally implements interface `ISimTimedComponent` to inherit the properties of a timed component. This way, the generated component classes for single-in components can act as both, incoming port and component.

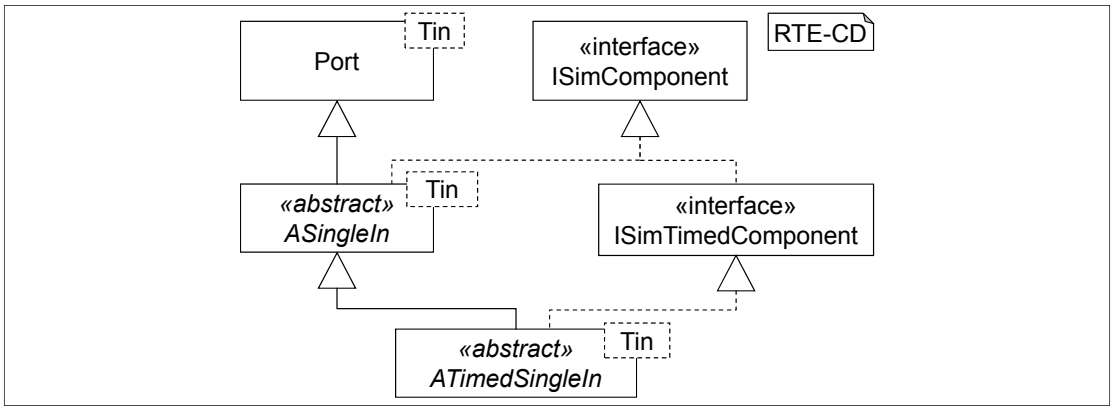


Figure 5.51: Extension of the RTE to support single-in components.

Figure 5.52 compares the generated code of a regular atomic component with the optimized code of a single-in atomic component. In the upper left corner, atomic component `DoorEval` is shown which has two incoming ports `doorStatus` and `switchStatus`. An excerpt of the generated class `ADoorEval` is shown in the top right corner. It extends the RTE class `ATimedComponent` since the default behavior class is timed (cf. Section 4.4). It has a private field for each incoming port and the corresponding getters return these fields.

The lower left corner of Figure 5.52 depicts single-in component `AlarmCheck` which has only one incoming port named `alarmStatus`. An excerpt of the resulting component class `AAlarmCheck` is given in the lower right corner. The used superclass is `ATimedSingleIn` parametrized to the port type `AlarmStatus`. In case of a single-in component, no field is generated for the incoming port `alarmStatus` and the corresponding getter returns the component itself (`this`).

Please note that this optimization only effects the generated code. Neither a component modeler who reuses a single-in component in a decomposed component type, nor the implementer of the component behavior of a single-in component has to deal with this optimizations. Since the concrete port class is determined by the inheritance hierarchy, this optimization heavily influences the testability of single-in components. For example, it is not possible to instrument a

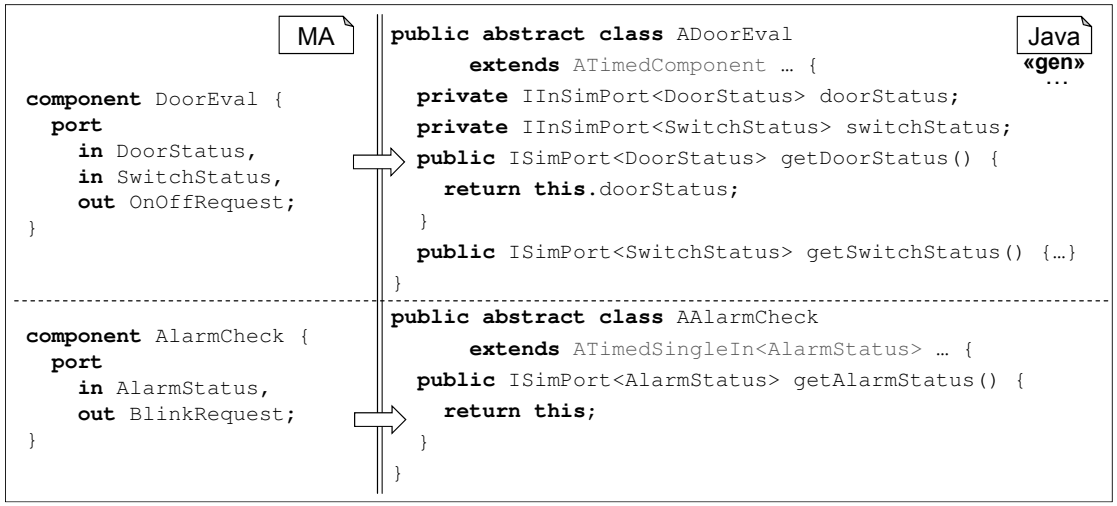


Figure 5.52: Optimization: Difference between the generated code for single-in and regular atomic components.

single-in component with test ports since component and port implementation are unified. The experience has shown that the resulting little reduction of allocated memory does not justify the absence of testability. Consequently, this optimization is disabled by default and has to be explicitly switched on. It can be useful if a simulation shall be executed in a very constrained environment. Otherwise, it is highly recommended to keep this optimization disabled. Especially, if a component library is to be developed.

5.6.2 Reduction of ForwardPorts in Decomposed Components

Actually, decomposed components do not produce behavior themselves and generated Java components do not need to instantiate dedicated port objects. If asked for an outgoing or incoming port (via a getter), they simply can return the outgoing or incoming port of the connected subcomponent. As explained in Section 5.4.3, this is only possible for outgoing ports since they can be connected to a single outgoing port of a subcomponent. Since incoming ports can be connected to more than one receiver, this optimization cannot be applied straight forward for incoming ports. For this reason, `ForwardPorts` are introduced which are instantiated for incoming ports of decomposed components. All incoming ports of the connected subcomponents are registered at these forwarding ports during the component setup. The `accept` method of these ports simply delegates the given message to the `accept` method of the connected ports.

The aim of this optimization is to reduce the amount of `ForwardPorts`. If only one target is connected to an incoming port, the same mechanism can be applied for incoming ports of a decomposed component, too. In this case, the corresponding getter for the incoming port simply returns the incoming port of the connected subcomponent. Hence, no `ForwardPort` object has to be created and the delegation step from forwarding to connected ports is omitted.

Figure 5.53 illustrates this optimization by comparing the previous with the optimized object

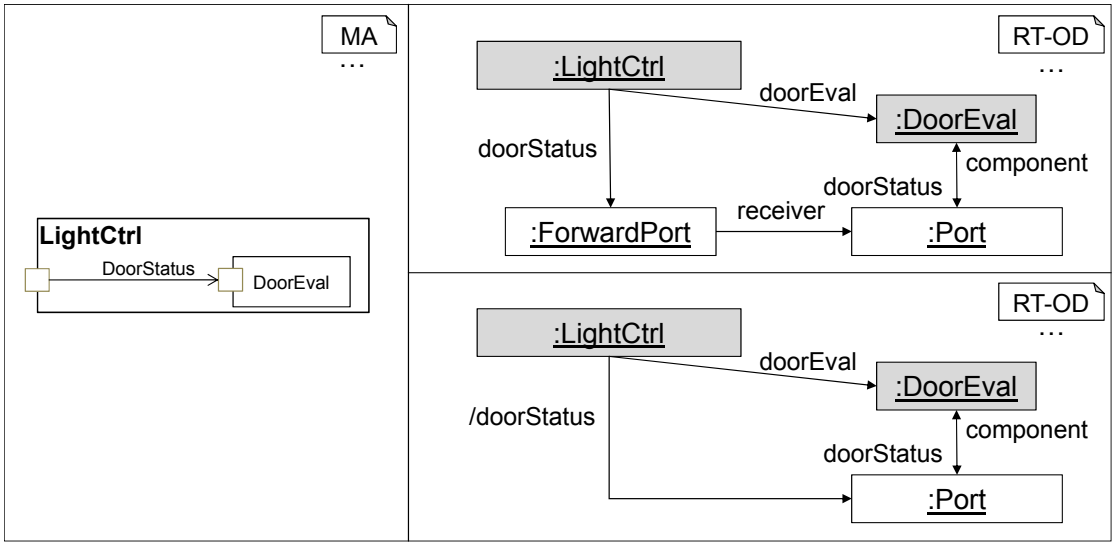


Figure 5.53: Optimization: Avoiding not needed ForwardPorts in decomposed components.

graph of component `LighCtrl`. On the left-hand side, an excerpt of the component definition is given. It can be seen that the incoming port `doorStatus` is exclusively connected to an incoming port of subcomponent `doorEval`. The object diagram in the top right corner depicts the unoptimized object graph which represents the marked connector and ports of the component. The `LighCtrl` object has a link `doorStatus` to a `ForwardPort` which is connected to the port object that itself is linked to the object that represents subcomponent `doorEval`. Since only a single receiver is registered at the `ForwardPort`, it can be omitted.

The bottom right corner contains the optimized object graph. It is shown that the `LightCtrl` object does not have a linked `ForwardPort` anymore. It is replaced by a derived link which directly points to the connected port object that belongs to subcomponent `doorEval`. On the level of the generated code the derived link is realized by the port getter. In the optimized version it returns the incoming port of its subcomponent instead of the private `ForwardPort` field.

5.6.3 Reuse of Tick Objects

As explained in Section 4.3.2, a component emits a tick on each outgoing port when it has received a tick on each incoming port. Beside their semantics, tick objects do not contain any further information or payload. Consequently, tick objects are reused in the simulation to avoid the overhead of tick object creation and ease tick comparison. Since ports in a MontiArc simulation are strictly typed using generics, ticks, just like messages, have to be typed to the type of the port which emits the tick. Thus, three solutions for the typesafe reuse of ticks are feasible:

1. A central rawtype tick object is stored in a constant which is used by all outgoing ports to emit ticks. Since the tick implementation is final and does not contain any generic payload, using a rawtype tick is safe at simulation runtime. Nevertheless, this solution generates compiler warnings since the rawtype tick needs unchecked conversation to conform to the

required type of the send method's argument. These warnings can be either ignored or suppressed each time the central tick is used. Both options are not recommended since either potentially upcoming new warnings are hidden by the high amount of ignored warnings, or the client code is cluttered with annotations that suppress the warnings.

2. To overcome the drawbacks of the first solution, a typed tick object for each used message type can be managed by a central tick factory. The ticks are stored in a map which maps each type name to the corresponding tick object. Components initially register a tick for each outgoing port at the factory. If a tick is needed, it can be obtained by passing the type name to a generic `get` method. While solving the type safety problem, some additional overhead is generated by the `get()` method of the internally used map which is expected to be executed in constant-time.
3. The implemented solution combines both, type safety without warnings in the client code while avoiding additional overhead. Therefore, the central rawtype tick object is stored in a private constant in class `Tick`. Further, a static generic method `<T> Tick<T> get()` is provided for the clients which returns the rawtype tick. Due to the generic type parameters of the method, the returned tick seem to be correctly typed for the client and the client code can be compiled without warnings.

By reusing a central tick object to represent the time flow on all channels in a simulation, the overall amount of simulation objects is extremely reduced.

5.7 MontiArc Tools

MontiArc provides several tools which can be used to develop MontiArc components. A basic command line interface (CLI) tool is presented in Section 5.7.1. It is internally used by the MontiArc Maven plugin which is described in Section 5.7.2. Finally, Section 5.7.3 presents the MontiArc integrated development environment (IDE) which integrates both other tools. In this way the functionality increases and becomes more powerful from tool to tool, while the concrete usage of the basic CLI tool becomes more transparent for the end user.

5.7.1 Command Line Interface

MontiArc's main Java class `MontiArcGeneratorTool` is controlled and configured using parameters which are passed to its constructor. It provides a static main method which can be used directly from a command line. The accepted parameters are depicted and explained in Table 5.54. Since MontiArc provides a default configuration, several parameters are optional. Thus, it is possible to use the aforementioned tool by passing the path to the models that are to be processed as a single parameter.

Parameter	Description
(*$directory* \| *$file*)	Input *$directory* or input *$file*. If a directory is used, all contained model files are processed. Alternatively, a model file may be directly passed to the tool. It is possible to pass multiple directories and/or model files to the tool.
`-mp` (*$directory* \| *$jarFile*)	Adds the given *$directory* or *$jarFile* to the modelpath of the tool. The modelpath is used to load referenced models, model elements, or further exported symbol table entries. This parameter can be used multiple times. The default modelpath contains the directory `src/main/models`.
`-symtabdir` *$directory*	Output directory for exported symbol table entries. If this parameter is omitted, the default symbol table directory `target/generated-sources/montiarc/symboltable` is used.
`-out` *$directory*	Output directory for generated files or further artifacts created during the tool execution. If this parameter is omitted, the default output directory `target/generated-sources/montiarc/sourcecode` is used.
`-analysis` *$modelKind* *$workflow*	Executes the workflow registered with id *$workflow* during the analysis phase of the tool execution. This workflow is executed on models with kind *$modelKind*. For example, `-analysis arc parse` will execute the parsing workflow on all processed MontiArc models. Model kind `ALL` can be used to execute the workflow for all processable model kinds. Multiple analysis workflows can be registered which are executed in the given order. If no analysis and synthesis (see below) workflows are explicitly given, the following default analysis workflows are registered: `parse` for all model kinds, `setname` and `addImports` for Java models, `init` and `createExported` for all model kinds. Please note that in this case, the default synthesis workflows are registered, too. A description of the mentioned workflows is given in Section 5.1.
`-synthesis` *$modelKind* *$workflow*	Executes the workflow registered with id *$workflow* during the synthesis phase of the tool execution. This workflow is executed on models with kind *$modelKind*. Multiple synthesis workflows can be registered which are executed in the given order. If no analysis (see above) and synthesis workflows are given, the following default synthesis workflows are registered together with the default analysis workflows:

Table 5.54 continued on next page

Parameter	Description
`-synthesis` *(continued)*	`prepareCheck` for all model kinds, `preCheckTransformation`, `check`, and `preCodegenTransformation` for MontiArc models. A description of these workflows is given in Section 5.1.
`-genlog`	This optional parameter enables logging within the generation process. Two kinds of artifacts are created. First, a textual protocol for each processed model and registered generator. It protocols which templates are called and which internal variables are set within the generation process. Second, for each processed model a GraphML (Graph Markup Language) [BEH+02] file is generated which can be visualized with, e.g., yEd[2]. The graph contains the source model, the called templates, and the produced artifacts. Please note, the layout of the contained graph has to be adjusted manually. Alternatively, yEd can automatically layout the graph (Tools → Fit Node to Label..., Layout → Hierarchical...).
`-generator` *$name $nodeClass*	Executes a generator workflow. If it discovers an AST node of class *$nodeClass*, it passes this node to generator *$name*. Node class and generator name have to be fully qualified. Nevertheless, `montiarccomponent` serves as an abbreviation for the AST node of the MontiArc compilation unit. The generator and its templates have to be available in the Java class path. A list of MontiArc generators located in package `mc.umlp.arc` is given in Table 5.55.

Table 5.54: Parameters of the `MontiArcGeneratorTool`.

Generator	Description
`ComponentInterfaceMain`	Generates Java interfaces for each component definition.
`ComponentFactoryMain`	Generates a Java factory for each component definition.
`ComponentMain`	Generates abstract superclasses for atomic components and implementations for decomposed components.
`ComponentSetupMain`	Generates a setup class for each component to simplify asynchronous usage of MontiArc components in Java classes. Such a setup offers methods to pass messages to the incoming ports of a component. The outgoing ports of the component trigger observers as a call-back which have to be registered for each outgoing port.

Table 5.55 continued on next page

[2] yEd website `http://www.yworks.com/en/products_yed_about.html`.

Generator	Description
`ComponentStubMain`	Generates a stub implementation for each decomposed component. Stubs can be used in white-box tests to replace the concrete implementation of decomposed components with stub implementations.
`SimMain`	Generates an interactive simulation for each processed component. Please note that this generator has been developed in a lab course and does not support all MontiArc features (e.g., generic components are not supported).

Table 5.55: Provided MontiArc generators in package `mc.umlp.arc`.

The list of available generators is given in Table 5.55. The default configuration of the `MontiArcGeneratorTool` contains the three generators `ComponentInterfaceMain`, `ComponentFactoryMain` and `ComponentMain`. Thus, if no concrete generators are registered using the `-generator` parameter (cf. Table 5.54), these three generators are executed for all processed MontiArc models.

5.7.2 MontiArc Maven Plugin

The MontiArc Maven Plugin provides means to use MontiArc within Maven builds. It uses the previously described command line tool (cf. Section 5.7.1) and provides all needed dependencies. This way, the user only has to adjust the configuration if the default configuration is not sufficient. The plugin provides four different goals which are bound to different phases of a Maven build. For an introduction to the Maven build lifecycle and its phases please refer to [www14j]. A description of all goals and their associated phases is given in Table 5.56. Available MontiArc specific configuration parameters are described in Table 5.57. Please note that the MontiArc Maven Plugin inherits further MontiCore specific configuration parameters from its base tooling. Please refer to [www12] for a documentation of these parameters.

Goal	Phase	Description
clean	clean	This goal deletes all files generated by previous plugin executions. Namely the generator output directory, the symbol table directory, and the modelpath configuration file (`ma.cfg`).
configure	initialize	Configures the current Maven project for the execution of MontiArc. In particular, the modelpath configuration file (`ma.cfg`) is generated by adding references to the artifacts which correspond to the project dependencies with classifiers `models`, `symbols`, `sources`, and `bootstrap`.

Table 5.56 continued on next page

Goal	Phase	Description
generate	generate-sources	Invokes the configured MontiArc generators. By default, interfaces, factories, and component classes are generated for each MontiArc component located in the models directory.
doc	prepare-package	Generates a HTML documentation similar to Java Doc for all components located in the models directory. By default, the documentation is generated to directory `target/madoc`.

Table 5.56: Goals of the MontiArc Maven Plugin and their target phases.

Parameter	Default Value	Description
buildIncrementally	true	Determines whether MontiArc should generate incrementally. If set to true, only models that have been changed since the last generation process are passed to the MontiArc tool.
checkCoCos	true	Determines whether MontiArc should check context conditions (see Section 3.5) for processed models. If these checks fail, code generation is skipped for the affected model.
countMetric	false	Determines whether MontiArc should execute a metric which counts elements of processed models. The results are exported to file `ComponentStatistics.txt` in the configured output directory.
generateCDTypes	true	Determines whether MontiArc should generate Java types from processed UML/P CDs which are used for port data type definitions. If CD types are used within a component, this parameter should not be switched to `false`. If so, the generated component classes cannot be compiled since referenced data types are missing.
generateComponentSetup	false	Determines whether MontiArc should generate setup classes using generator `ComponentSetupMain` (cf. Table 5.55).
generateStubs	false	Determines whether MontiArc should generate stub superclasses using generator `ComponentStubMain` (cf. Table 5.55).

Table 5.57 continued on next page

Parameter	Default Value	Description
generateInteractive-Simulation	false	Determines whether MontiArc should generate a simple interactive component simulation using generator `SimMain` (cf. Table 5.55).
modelDirectory	`src/main /models`	Sets the model input directory.
docOutputDirectory	`target/madoc`	Sets the target directory for generated component documentation.
generateProtocol	false	Determines whether log files have to be generated (see Table 5.54, `-genlog`).
optimizeSingleIn	false	Determines if components with a single incoming port should be optimized by merging the port and the component into a single object.
optimizeSingleIn *(continued)*	false	Please be aware that this heavily influences testability of affected components since ports cannot be replaced by test ports anymore (see Section 5.6.1).
docVersion	${project.version}	Version number respectively text which is used for `@version` tags (cf. Section 6.7.2) in the generated documentation.
javaDocUrls	Java 7 API doc	A list of URLs that are used to resolve Java API documentations of types which are referenced with `@link` tags (cf. Section 6.7.2).
isPublic	true	Configures goal *doc* to generate a documentation of model elements with public visibility. If set to false, protected elements, e.g., components and connectors, are also considered for the generated documentation.

Table 5.57: Configuration parameters of the MontiArc Maven Plugin.

5.7.3 Eclipse IDE

The MontiArc Eclipse IDE provides an editor and further functionality to create, modify, and test MontiArc components. Additionally, it integrates the previously described MontiArc Maven Plugin into Eclipse. To ease modeling with MontiArc, the MontiArc IDE offers the following functionalities:

- **New project wizard** for MontiArc projects. Based on a given name it creates an Eclipse project which already contains the default folder structure and a default Maven configuration. Projects created with this wizard are configured to reuse the default MontiArc libraries (cf. Section 6.8.2) and to process I/O-Test-models (see Section 6.4) within manually triggered Maven builds.

- **New model wizard** for MontiArc and I/O-Test models. Creates empty components or test models.
- **Text editors** for MontiArc and I/O-Test models.
- **Syntax highlighting** of language keywords to ease textual modeling.
- **Outline** that displays the structure of the currently opened model to gain an overview in large models.
- **Problem reports** that display failed context condition checks. Associates **markers** point to the model element that injured the context condition. In this way, defective models are easier to repair.
- Supports **active specification** by context sensitive autocompletion to prevent creating defective models:
  - Offers available component or data types to easily reference library components as subcomponents.
  - Helps to correctly connect ports. Only valid targets, which are compatible with the current connector source, are offered.
  - Offers suitable model elements and keywords depending on the current model state.
  - Helps to implement I/O-Test models by creating suitable test templates for the current component under test.
- **Integrated documentation** of referenced models available via autocompletion or mouse hover. The documented functionality of referenced models can be used without having access to the model's sources.
- Integration of the MontiArc Maven Plugin into **Eclipse builds**. If automatic builds are activated, the IDE automatically keeps generated artifacts of changed MontiArc models up to date. Changes are directly reflected in the generated code.

A screenshot of the MontiArc IDE that demonstrates some of these features is given in Figure 5.58. It can be seen that the currently opened component model `ABP` produces two warnings. The affected model elements are marked with a warning symbol and a corresponding problem report is shown in the `Problems` view. The active autocompletion offers two distinct component types for the already typed prefix "`Los`". Further, it provides documentation of the currently selected type `LossyDelayedChannel`. On the right-hand side, the outline gives an overview of the component's structure.

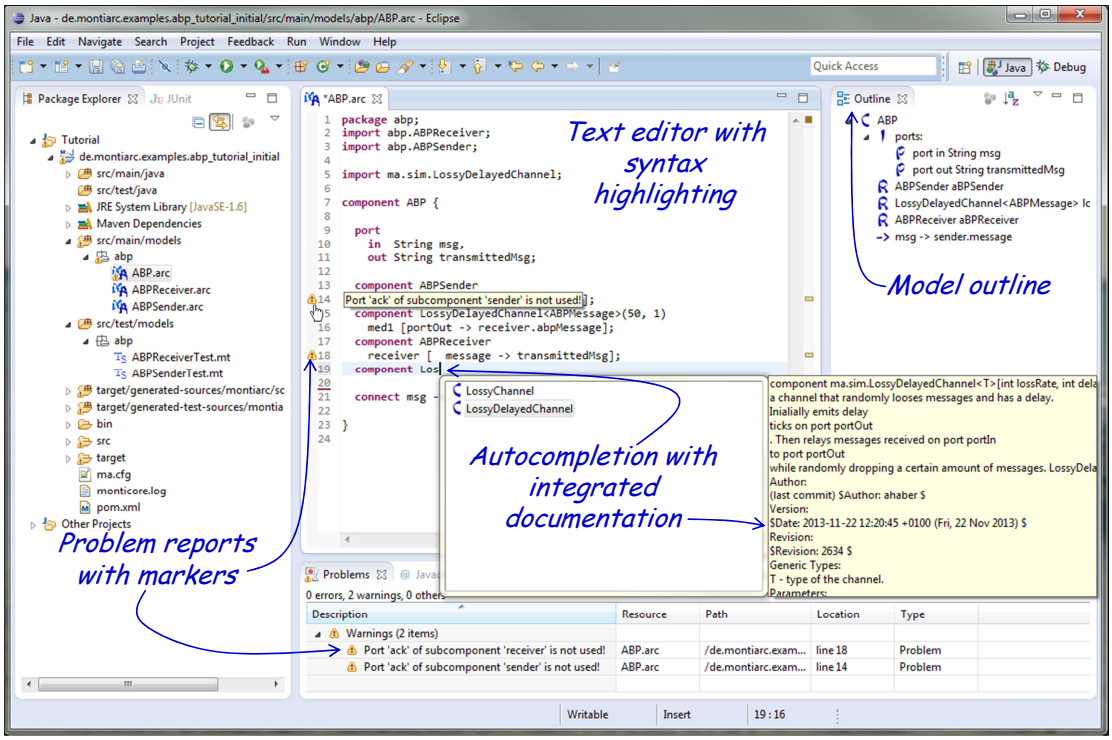


Figure 5.58: MontiArc Eclipse IDE.

Chapter 6

Tutorial: Development and Simulation of MontiArc Components

The MontiArc architecture description language (ADL) presented in Chapter 3 is used to model distributed, interactive systems. Using the corresponding FOCUS simulation, one is able to rapidly prototype, test, and explore the modeled system. MontiArc comes with an integrated development environment (IDE) (see Section 5.7) that supports the modeler with integrated context condition checks and active specification to simplify modeling. The contained build tools support continuous and distributed component development. This tutorial presents how to use the tools to model and implement MontiArc components. A basic knowledge about MontiArc and Java is required. The following aspects of MontiArc development are covered and demonstrated:

- **Setup**: How to get started and set up the MontiArc IDE is presented in Section 6.1.
- **Modeling**: Modeling of atomic and decomposed MontiArc components is demonstrated using the running example of the alternating bit protocol in Section 6.2.
- **Behavior implementation**: The implementation of atomic component behavior is presented in Section 6.3.
- **Testing**: How to write and execute black- and white-box tests for MontiArc components is shown in Section 6.4.
- **Generalization**: Means of the MontiArc language that increase reusability of components are demonstrated in Section 6.5.
- **Optimization Tests**: Techniques to test and compare different system configurations are described in Section 6.6.
- **Documentation**: How to document MontiArc components to simplify and support their reuse is shown in Section 6.7.

Beside this main path of the tutorial, which follows the consistent development of MontiArc components, this chapter contains additional background information for the interested reader or advanced MontiArc modeler. To visually separate corresponding sections from the main path of the tutorial, a * is appended to the section names. These sections can be skipped if simply following the story of the tutorial. This mainly affects:

- Section 6.8 that depicts the structure of MontiArc libraries and describes how to use and create them, and
- Section 6.9 that explains how to physically distribute a MontiArc simulation on distinct nodes.

6.1 Getting Started

This tutorial is based on the MontiArc Eclipse IDE which is presented in Section 5.7.3. It is recommended to download a fresh Eclipse that already contains a Maven integration (m2e) from `http://www.eclipse.org/downloads/`[1] and extract it to a desired directory. If an existing Eclipse is to be used, please make sure that m2e is installed in a version >= 1.4.0.

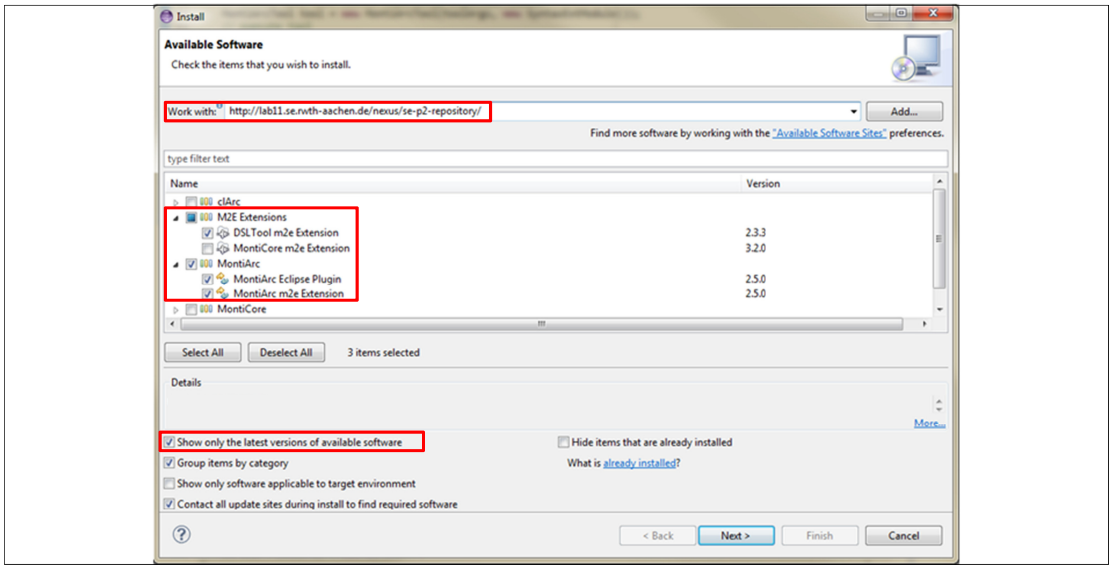


Figure 6.1: Install MontiArc in Eclipse using the MontiArc update site.

The MontiArc Eclipse tooling may be installed either via update site or manually. The installation using the update site is recommended since updates of the MontiArc tooling can be automatically acquired. Thus, we perform the following steps to install MontiArc using the update site:

1. Start Eclipse.
2. Open the help menu and press "Install New Software...". The resulting dialogue is shown in Figure 6.1.
3. Insert the URL of the update site[2] into the "Work with" field. Select the **MontiArc Eclipse Plugin** and the **MontiArc m2e Extension** in version 2.5.0 as well as the **DSLTool m2e Extension** as depicted in Figure 6.1. The MontiArc plugin provides the MontiArc IDE presented in Section 5.7.3, the m2e extensions handle the integration of the montiarc-maven-plugin (see Section 5.7.2) respectively the dsltool-maven-plugin into Eclipse builds. Please note that this tutorial is tested with MontiArc 2.5.0. Thus, it is recommended to install this version. If a newer version is displayed in the installation dialogue,

[1] This tutorial is tested with Eclipse IDE for Java Developers Version: Kepler Service Release 1 and Eclipse Standard/SDK Version: Luna Service Release 1 (4.4.1).

[2] A reference to the update site and the standalone installation can be found here: `http://www.monticore.de/languages/montiarc/download/`.

the checkbox “Show only the latest versions of available software” has to be unchecked.

4. Press “Next” twice, accept the licence agreement, press finish and restart Eclipse.

To manually install MontiArc, download the MontiArc archive from the website[2] and extract it into the `plugins` or `dropins` sub-directory of your Eclipse installation. Then restart Eclipse with argument `-clean` to refresh its plugin registry.

To check whether MontiArc is installed and Eclipse is set up correctly, create a new MontiArc Project. For this, select the “File” menu, select “New” and “Other”. Then select “MontiArc-Project” in the wizard and press “Next” to insert a project name, e.g., “HelloMontiArc”, and press “Finish”. The project structure of the created project is depicted on the left side of Figure 6.2 and is based on common Maven conventions. Further information about the project structure is given in the contained `readme.txt` file. Component models should be created in directory “src/main/models”. Create a new package, e.g., “ma.hello” in this folder, then right-click the package, select “New”, “Other” and “MontiArc-Component”. After entering a component name, e.g., “HelloMontiArc”, select “Finish”. The created component should be opened in the MontiArc editor afterwards.

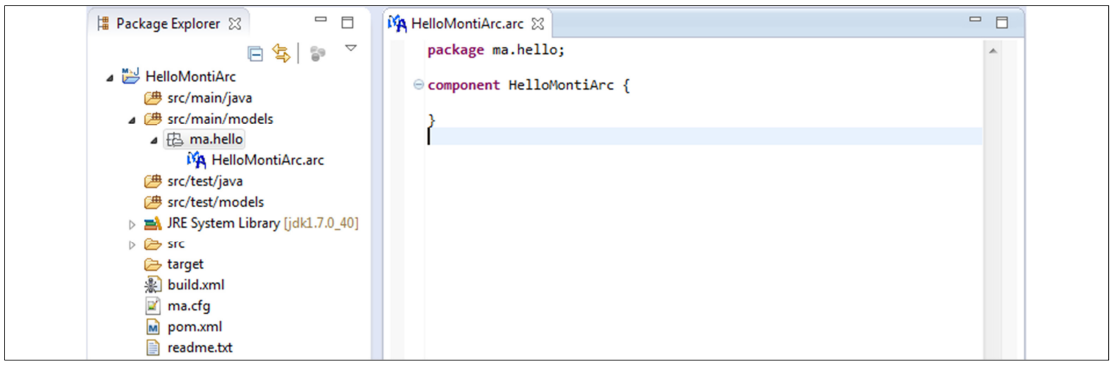


Figure 6.2: A MontiArc Eclipse project.

To test the Maven setup, right-click on the project, select “Run As” and “Maven install”. During the initial execution, Maven automatically downloads MontiArc, its dependencies, and needed Maven plugins to store them in the local Maven repository. Update the project by right-clicking it, select “Maven” and “Update Project”, then press “OK”. If the build does not succeed or the project is still marked with errors after an update, please check the MontiArc FAQ[3].

6.2 Illustrative Example - Alternating Bit Protocol

We continue by creating an architectural models using MontiArc. For this purpose we initially introduce the Alternating Bit Protocol (ABP) as a running example which is used within this chapter. After discussing its requirements, we explain how to set up the example and how to model the contained protocol components with the MontiArc IDE.

[3]MontiArc FAQ, `http://www.monticore.de/languages/montiarc/faq/`

The ABP is a very simple data link layer protocol which ensures lossless transmission of messages over an unreliable medium [www14d]. It mainly consists of a sender and a receiver component that are connected as depicted in Figure 6.3. The `sender` combines a received message with an acknowledgement bit to an `ABPMessage` which is transmitted via the lossy channel `med1` to the `receiver`. The receiver then forwards the contained message to port `transmittedMsg` and sends the contained bit back to the sender via lossy channel `med2`. If this bit is equivalent to the last sent bit, the sender will continue transmitting the next message with an inverted bit. If the sender does not receive an acknowledgement bit within a defined time span or the received bit differs from the last sent bit, the message has been lost. Thus, the sender transmits the last message again.

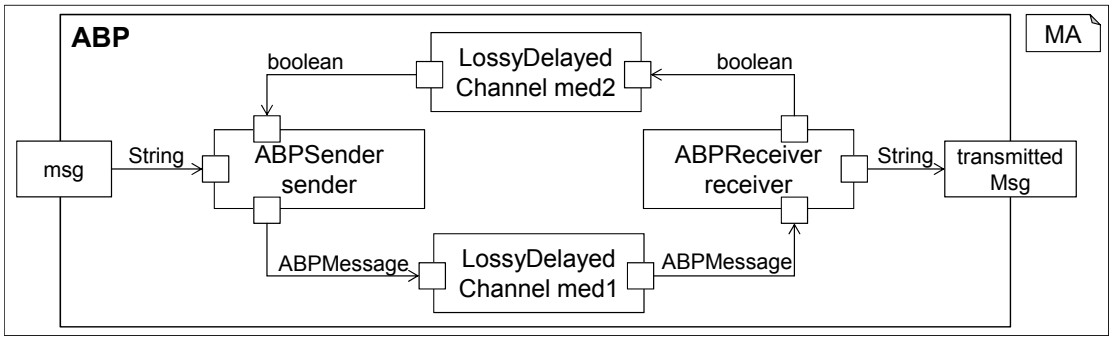


Figure 6.3: Alternating Bit Protocol component model.

6.2.1 Requirements

The behavior of the ABP is specified by the following requirements. It is distinguished between sender and receiver requirements that either specify the behavior of the sender part or the receiver part of the ABP.

ABPSender Requirements

ABP_S1 The sender has to store the newest transmission flag. It is alternated for each message that has to be transmitted. The initial transmission flag is `true`.

ABP_S2 A sender has a message buffer realized as a FIFO queue.

ABP_S3 The sender has two states:

- **ABP_S3.1** `RDY`: Sender is ready to transmit data. The message buffer is empty in this state.
- **ABP_S3.2** `W8ING`: Sender is waiting for an acknowledgment of transmitted data. The message buffer contains one or more messages in this state.

ABP_S4 The initial state of the sender is `RDY`.

ABP_S5 If the sender is in state `RDY`:

- **ABP_S5.1** ... and it receives a message, the content of the message is encapsulated into an `ABPMessage` object together with the current transmission flag. This object is then emitted and a timeout is started that lasts three time intervals. The message

is preventively stored in the buffer in case it needs to be resend. Then, the sender switches to state `W8ING`.

ABP_S5.2 ... and it receives an acknowledgment it is ignored.

ABP_S6 If the sender is in state `W8ING`:

ABP_S6.1 ... and receives a message, the message is buffered.

ABP_S6.2 ... and receives an acknowledgment that matches the current transmission flag, the transmission flag is inverted and the transmitted message is removed from the buffer. If existing, the next buffered message is transmitted (see *ABP_S5.1*) and the timeout timer is set to three time intervals. If the buffer does not contain any further elements, the sender switches to state `RDY` and the timeout timer has to be deactivated.

ABP_S6.3 ... and receives an acknowledgment that does not match the current transmission flag, the sender has to resent the last message that is stored in the buffer (see *ABP_S5.1*).

ABP_S6.4 ... and does not receive an acknowledgment for a period of three time intervals, the sender has to resent the last message that is stored in the buffer (see *ABP_S5.1*).

ABPReceiver Requirements

ABP_R1 The receiver has to store the transmission flag of the latest transmitted ABPMessage.

ABP_R2 This locally stored flag is initialized with `false`. Thus, the receiver initially expects a transmission flag with the value `true`.

ABP_R3 If an ABPMessage with an "ack" value (the value of the alternating bit) that differs from the locally stored transmission flag is received, the last transmission has been successful. Thus, the received flag has to be stored and sent back to the sender. Additionally, the payload of the ABPMessage is emitted.

ABP_R4 If an ABPMessage with an "ack" value equal to the stored transmission flag is received, the last transmission has not been successful. Thus, to signal the sender that the last transmission failed, the stored flag is to be resent.

6.2.2 Example Setup

The following steps help to set up a common base for the `ABP` example:

1. Download example "Tutorial: Alternating Bit Protocol" from the examples section[4] in version 2.5.0 from the MontiArc website.
2. Import the example into the Eclipse workspace by selecting "File" and "Import".
3. Select "General" and "Existing Projects into Workspace" and press "Next" .
4. Perform the following steps in the import wizard that are also depicted in Figure 6.4:
   a) Toggle "Select archive file",
   b) press "Browse" and then open the downloaded zip archive in the file browser. The project that is to be imported should be selected automatically.

[4] MontiArc examples, `http://www.monticore.de/languages/montiarc/examples/`.

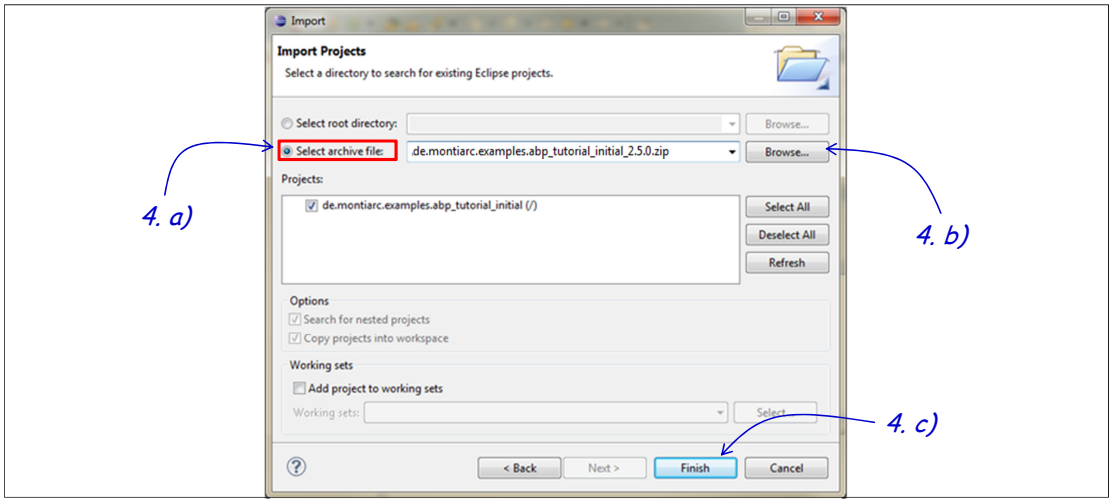


Figure 6.4: Import the initial example project.

c) Then press “Finish”.

5. Update the created project as depicted in Figure 6.5 by right-clicking the project, selecting “Maven” and “Update Project. . . ”.
6. Finally perform a Maven install by right-clicking the project and selecting “Run as” and “Maven install”.

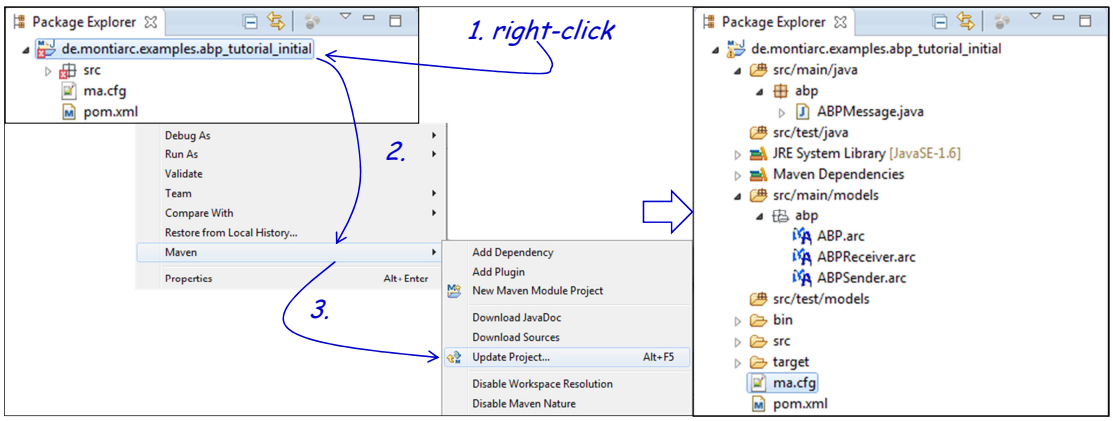


Figure 6.5: Updating a Maven project.

6.2.3 Modeling

If the imported project does not contain any errors and the executed Maven build executes successfully, you can start detailing the contained component models. As depicted in Figure 6.3, the component `ABP` (located in package `abp` within the directory `src/main/models`) needs

a port to accept and a port to deliver transmitted messages. Thus, we add the ports `msg` and `transmittedMsg` to component `ABP` (see Listing 6.6).

```
1  port                                                    MA
2    in  String msg,                                       ...
3    out String transmittedMsg;
```

Listing 6.6: The port interface of component `ABP`.

Then, we model the interface of component `ABPSender` that is located in the same package. It needs a port to accept String data messages (`message`), a port to accept an acknowledgment bit coded as a `Boolean` value from the receiver (`ack`), and a port to transmit alternating bit messages `abpMessage`. Please note that the example project already contains a data type definition `ABPMessage` which is used as data type for the latter port. It is located in package `abp` in the Java source folder and envelops a data message together with an acknowledgment bit. The resulting interface of component `ABPSender` is depicted in Listing 6.7.

```
1  port                                                    MA
2    in String message,                                    ...
3    in Boolean ack,
4    out ABPMessage abpMessage;
```

Listing 6.7: `ABPSender` component port interface.

Afterwards, we define the interface of component `ABPReceiver`. It has to accept alternating bit messages (`abpMessage`) and transmit the contained data via port `message`. It also has to send back the acknowledgment bit to the sender (`ack`). Since the component does not need any time information, we select the `untimed` timing domain. The expected component interface is depicted in Listing 6.8.

```
1  timing untimed;                                         MA
2  port                                                    ...
3    in ABPMessage abpMessage,
4    out Boolean ack,
5    out String message;
```

Listing 6.8: `ABPReceiver` component port interface and timing domain.

Now, we are ready to implement the internal structure of component `ABP` by decomposing it. For this, we declare some subcomponents and add connectors which connect the interfaces of the subcomponents. Subcomponents start with the keyword `component` followed by the instantiated component type and an optional name. As depicted in Figure 6.9, the autocompletion (activated by pressing Ctrl + space) helps by offering available component types. Utilizing this feature, we add the following subcomponents to component `ABP`:

- A subcomponent `sender` with type `ABPSender` (its interface has been defined above).

- Another lossy delayed channel (component type `LossyDelayedChannel`) named `med2`. As suggested by the autocompletion, this component type is generic and parameterizable. Similar to `med1`, we parametrize `med2` by assigning `Boolean` to type parameter `T`. We further configure the component with a loss rate of 50 percent and a delay of one time interval. The loss rate is deterministically determined by the passed `ControlledRandom` object which is configured with the String `"1010"`. It is interpreted modulo wise to determine whether a received message is to be lost. A `1` means that the message is transmitted, a `0` means that the message is lost. In this way, the 1^{st}, 3^{rd}, 5^{th}, ... messages are transmitted and the 2^{nd}, 4^{th}, 6^{th}, ... messages are lost.
- A subcomponent named `receiver` with type `ABPReceiver`.

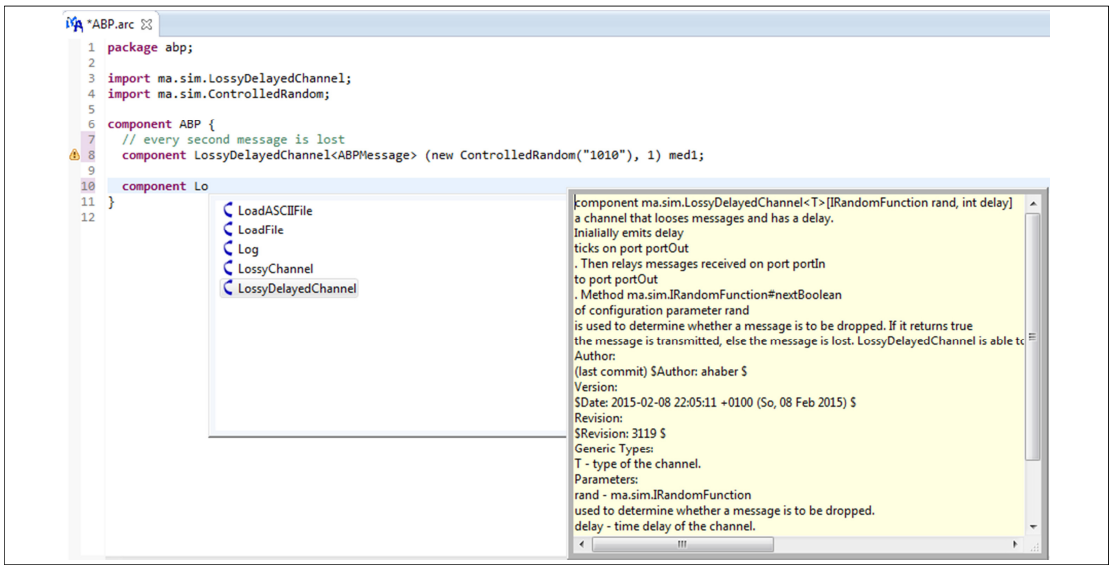


Figure 6.9: Editor autocompletion to add subcomponents.

The editor will now display some warnings about unused ports of component `ABP` and its subcomponents. To fix these issues, we connect the contained subcomponents with each other and the outer interface of the component. You may either uses simple connectors (see Section 3.4.3) which are directly attached to the name of a subcomponent or regular connectors. The following ports have to be connected:

- Port `msg` with port `message` from subcomponent `sender`,
- port `abpMessage` from subcomponent `sender` with port `portIn` from subcomponent `med1`,
- port `portOut` from subcomponent `med1` with port `abpMessage` from subcomponent `receiver`,
- port `ack` from subcomponent `receiver` with port `portIn` from subcomponent `med2`,
- port `portOut` from subcomponent `med2` with port `ack` from subcomponent `sender`,
- port `message` from subcomponent `receiver` with port `transmittedMsg`.

The resulting decomposition, which uses a mixture of simple (cf. ll. 2, 6, 9f and 14) and

regular connectors (cf. l. 16), is depicted in Listing 6.10.

```
1  component ABPSender                                                    MA
2    sender [abpMessage -> med1.portIn];                                  ...
3
4  component LossyDelayedChannel<ABPMessage>
5    (new ControlledRandom("1010"), 1) // lose every 2nd message
6    med1 [portOut -> receiver.abpMessage];
7
8  component ABPReceiver
9     receiver [ack -> med2.portIn;
10               message -> transmittedMsg];
11
12 component LossyDelayedChannel<Boolean>
13   (new ControlledRandom("1010"), 1)
14   med2 [portOut -> sender.ack];
15
16 connect msg -> sender.message;
```

Listing 6.10: Subcomponents and connectors of the ABP component.

According to requirement *ABP_S5.1*, the sender has to resent messages after a timeout of three time intervals. Consequently, we decompose component `ABPSender` the following way:

1. We add an inner component definition `ABPInnerSender` instantiated as subcomponent `sender`. This component implements the behavior of the ABP sender. Thus, on the one hand it handles all accepted messages from its surrounding component `ABPSender`. So it has to provide the same ports `messsage`, `ack`, and `abpMessage` with the same data types and directions. And, on the other hand, it controls the timer and reacts to timer events. Thus, we add an outgoing port `setTimer` to set the timer with an `Integer` value and an incoming port `timerEvent` with data type `Boolean` to get notified with timeouts. The resulting inner component definition is depicted in Listing 6.11. The interface has been split into two port groups to emphasize the relation of the grouped ports.
2. Add a subcomponent with library type `Timer` which realizes timeouts.
3. By adding an `autoconnect port` statement to component `ABPSender`, we automatically connect most of the ports.
4. The warning markers inform us, that the ports `trigger` from subcomponent `timer` and `timerEvent` from subcomponent `sender` are not connected yet. So we additionally connect these ports by creating a corresponding connector.
5. Now, another warning marker informs about a feedback cycle between subcomponents `sender` and `timer`. To break this cycle, we add a subcomponent which instantiates the library type `FixDelay`. Then, we change the target from the above created connector to the incoming port of the delay subcomponent and let the delay subcomponent sent to the previous target `sender.timerEvent`. An excerpt which depicts these connections is given in Listing 6.12.

To ease the following behavior implementation, we now generate Java components by running

```
component ABPInnerSender sender {
  port
    in String message,
    in Boolean ack,
    out ABPMessage abpMessage;

  port
    out Integer setTimer,
    in Boolean timerEvent;
}
```

MA
...

Listing 6.11: The interface of ABPSender's inner component ABPInnerSender.

```
component FixDelay<Boolean>(1) delay;
component Timer;

connect timer.trigger -> delay.portIn;
connect delay.portOut -> sender.timerEvent;
```

MA
...

Listing 6.12: Further subcomponents of component ABPSender and their connections.

`Maven install` and update the Maven project as depicted in Figure 6.5. A screenshot of a running MontiArc Maven plugin is depicted in Figure 6.13.

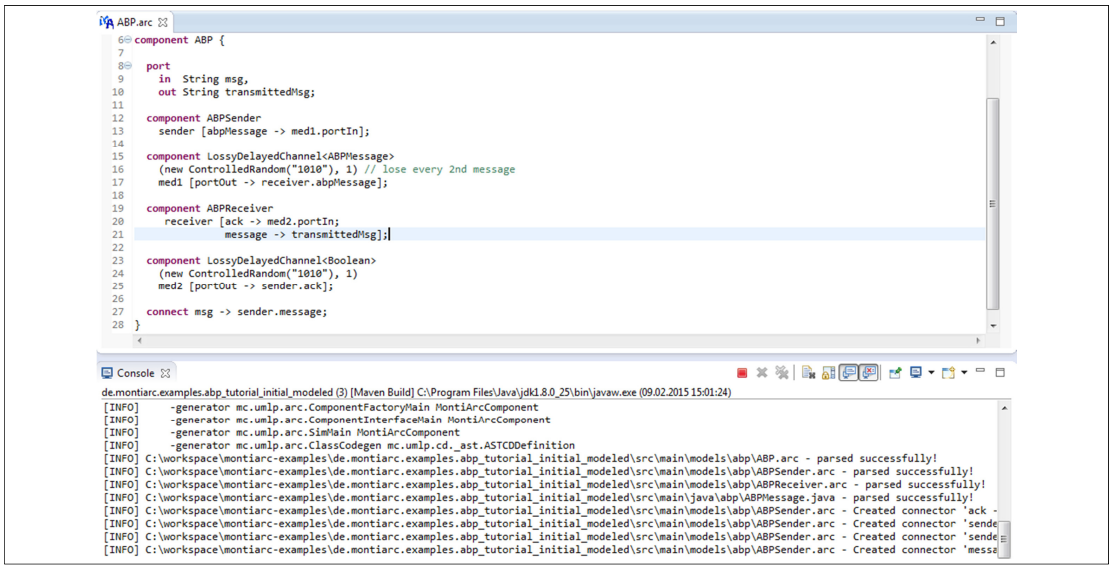


Figure 6.13: A running MontiArc Maven plugin.

6.3 Behavior Implementation

The behavior of MontiArc components is either generated by the composition of interacting subcomponents or by the implementation of atomic components. Since the basic MontiArc language does not provide means to directly realize the behavior within an atomic component, it has to be implemented externally. The default implementation language is Java. Nevertheless, it is also possible to create implementations in native languages such as C.

6.3.1 Behavior Implementation in Java

We continue by implementing the behavior of the atomic (inner) component definitions `ABPInnerSender` and `ABPReceiver`. For this, we create the package `abp` in the Java source directory `src/main/java` (see Figure 6.2) and create the Java class `ABPInnerSenderImpl` in this package. We use the generated class `AABPSender_ABPInnerSender` as superclass and add all unimplemented methods as depicted in Figure 6.14. Subsequently, we implement the sender requirements described in Section 6.2.1. The method `treatMessage` has to handle incoming messages, the method `treatAck` handles incoming acknowledgements. The sender states `RDY` and `W8ING` given in the requirements can be implicitly implemented by analyzing the state of the required message buffer. Please also consider the documentation of component type `Timer`. It is displayed when hovering with the mouse over the component type. Alternatively, the library documentation can be viewed online on the MontiArc website [5]. Additionally, consider the delay of one time interval which has been introduced by adding a `FixDelay` component into the feedback cycle. An exemplary implementation of the sender is given in the appendix in Listing E.1 on page 317. The contained sourcecode comments document how the given requirements are realized in the implementation.

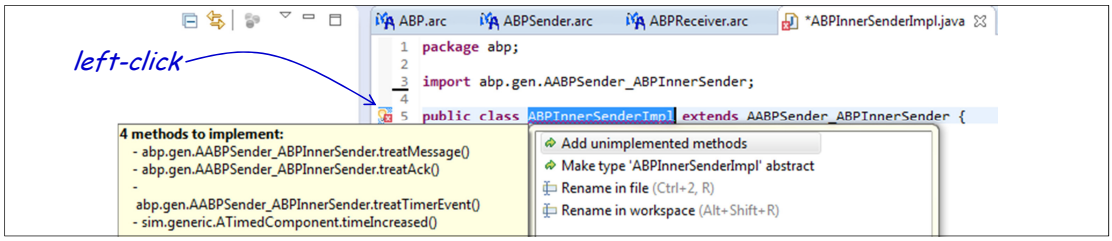


Figure 6.14: Quick fix to add missing event methods.

Now, create an additional Java class `ABPReceiverImpl` to implement the behavior of atomic component `ABPReceiver`. Use superclass `AABPReceiver` and add the unimplemented method as explained above. Again, we implement the class straight forward considering the receiver requirements described in Section 6.2.1. An exemplary implementation, which also documents how these requirements are realized, is given in Listing E.2 on page 319.

Finally, we regenerate the complete project with `Maven clean install`. This is needed since the factory-generator scans the project for existing behavior implementations which do

[5] MontiArc Component Library Documentation, `http://www.monticore.de/languages/montiarc/lib/`.

match the naming conventions (see Section 5.5.2 on page 165). Thus, the component factories have to be regenerated.

The generated system can now be interactively explored by starting the generated class `ABPRunSimulation`[6]. The generated simulator is depicted in Figure 6.15. It can be seen, that the `ABP` component receives three messages `First`, `Second`, and `Third` in time interval 43. The message `First` is emitted by output port `transmittedMsg` in time interval 58. This generated simulation can be used to interactively explore the interface behavior of the modeled components. Please note that the printed time stamps denote the current time of the simulated component and not the current time of a certain port. Because component `ABP` is a decomposed component, its current time is calculated by taking the minimum time of its contained subcomponents.

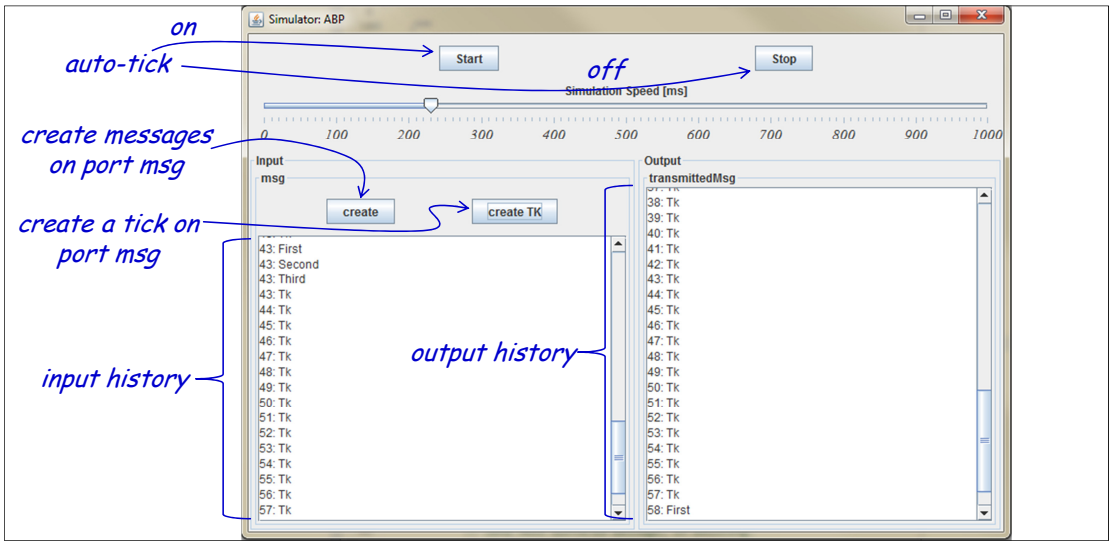


Figure 6.15: Interactive ABP simulation.

6.3.2 Native Behavior Implementation*

Sometimes, the behavior implementation of an atomic component already exists as a native C or C++ implementation, parts of the behavior need access to functionality provided by a native library, or the target implementation is to be realized in C anyway. Nevertheless, the simulation of such native atomic components and their interaction with their environment is desirable.

This subsection demonstrates how to integrate native code written in C into an atomic MontiArc component based on the simple example project "Simulation of Native C Components" which can be downloaded from the MontiArc website[7]. It can be imported into Eclipse as described in Section 6.2.2. Please consider the contained `readme.txt` file which documents

[6] Class `ABPRunSimulation` is located in directory `target/generated-sources/montiarc/source-code` in package `abp.gen`.

[7] MontiArc Examples, `http://www.monticore.de/languages/montiarc/examples/`.

how to setup the C compiler needed to compile the contained C sources.

The example project contains an atomic component `NativeAdder` which is depicted in Listing 6.16. This stateless synchronous component should compute the sum of the values received on both input ports and emit it via port `sum`. The concrete computation is realized in C as given in Listing 6.17.

MA

```
1 component NativeAdder {
2   timing sync;
3   port
4     in Integer addend1,
5     in Integer addend2,
6     out Integer sum;
7 }
```

Listing 6.16: Atomic component `NativeAdder` that is implemented in native C.

C
«handcoded»

```
1 #include <stdio.h>
2 #include "NativeImpl.h"
3
4 int add(int addend1, int addend2) {
5     return (addend1 + addend2);
6 }
```

Listing 6.17: Native implementation of add functionality.

Integrating C Code

Figure 6.18 depicts the used pattern which allows to integrate the given C implementation into the atomic component using the Java Native Interface (JNI). The regular Java behavior implementation is located in class `NativeAdderImpl` which has the generated superclass `ANativeAdder`. For this purpose it has to implement the method `timeStep`. Due to the used behavior class *synchronous* (see Section 4.4.4), missing values of the *data event tuple* are represented as `null`. Since `null` references cannot be easily handled by native implementations and `int` parameters are expected, missing values have to be mapped to a natural number, e.g., zero. An exemplary implementation of this mapping is depicted in Listing 6.19.

The lifted values are then passed as arguments to the static native method `nativeAdd` located in class `NativeWrapper`. This method is static to emphasize the static property of the linked native C implementation. This class is additionally responsible for loading the dynamic library `nativeImpl-x86_64.dll` which contains the C implementation to use. Encapsulating native methods within a distinct Java class that does not have a superclass eases JNI header generation using `javah`. In this case, `javah` does not have to consider dependencies and methods defined in the superclass. For that reason, the native method is declared in class `NativeWrapper` and not in the Java behavior implementation `NativeAdderImpl`.

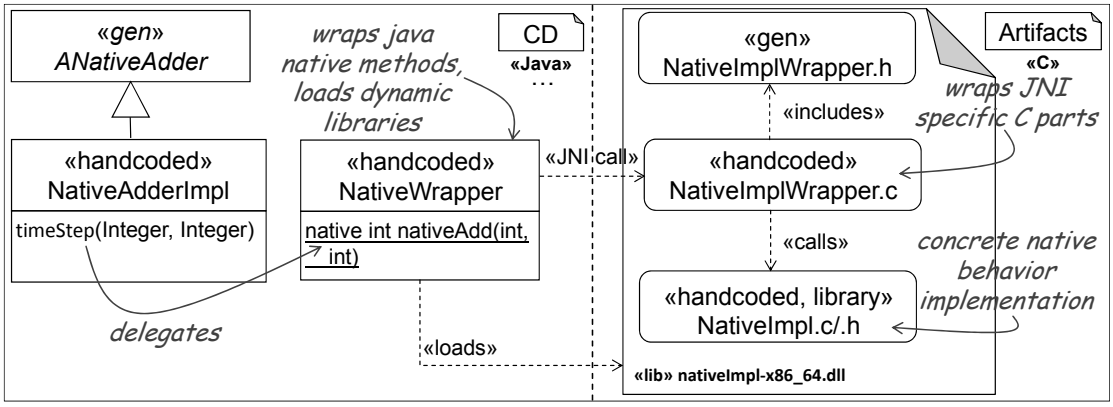


Figure 6.18: Pattern to integrate existing C code into atomic MontiArc components.

```
1  @Override                                                    Java
2  protected void timeStep(Integer add1, Integer add2) {       «handcoded»
3    // map missing values to zero
4    int a1 = (add1 != null ? add1 : 0);
5    int a2 = (add2 != null ? add2 : 0);
6    // call the native implementation
7    int sum = NativeWrapper.nativeAdd(a1, a2);
8    sendSum(sum);
9  }
```

Listing 6.19: Mapping from null to zero and computation delegation to the native method in class `NativeAdderImpl`.

To separate JNI specific parts from plain C code, a similar pattern is followed on the C side of the behavior implementation. The header file for the given `NativeImplWrapper` is generated by `javah` which uses Java class `NativeWrapper` as source. The concrete implementation is depicted in Listing 6.20. It is recommended to wrap JNI specific elements into dedicated wrappers for the following reasons. First, existing C libraries that should be used to realize component behavior are not linked against the needed JNI libraries (`jni.h`, l. 5) and their APIs have to be wrapped anyway. Second, newly created C code which realizes component behavior should not include the needed JNI libraries since these may not be available on the target platform. For this reason, all JNI specific C parts are encapsulated in the artifact `NativeImplWrapper.c` which delegates to the concrete behavior implementation given in artifact `NativeImpl.c`.

Adjusting the Project Build

To seamlessly integrate C code with the described pattern into the MontiArc build (see Section 5.7.2), the following two build steps have to be added:

1. **Header file generation:** The header files for the native implementation wrapper have to be generated using `javah`. This has to be done after MontiArc source code genera-

C «handcoded»

```c
#include <stdio.h>
#include <stdlib.h>
#include "../../../target/NativeImplWrapper.h"
#include "NativeImpl.h"
#include <jni.h>

JNIEXPORT jint JNICALL
    Java_de_montiarc_examples_natives_NativeWrapper_nativeAdd
  (JNIEnv *env, jclass clazz, jint a1, jint a2) {
  // delegates to the native implementation given in NativeImpl.c
  return add(a1, a2);
}
```

Listing 6.20: Wrap JNI specific elements within a dedicated wrapper.

tion and after Java source code compilation. The needed functionality is provided by the `jni-headers-maven-plugin` which is configured to be executed in the `process-classes` build phase.

2. **Compiling C code:** The contained C code has to be compiled. This has to be performed after the first step because the generated header files are included in the handwritten C artifacts and have to be compiled, too. Since the produced dynamic library is loaded at simulation runtime, the C code has to be compiled before tests are executed. In the given example project, C code is compiled using the `exec-maven-plugin` which calls gcc[8] with the needed arguments. The plugin is configured to be executed in the `process-classes` build phase after the `jni-headers-maven-plugin`.

Discussion

Using native implementations written in C or C++ to realize the behavior of atomic components introduces some restrictions which influence portability between distinct platforms on which the simulation is to be executed. Since native code, in contrast to Java, has to be compiled for a specific target platform, the binary code cannot be executed on an arbitrary platform. For example, it has to be explicitly compiled for Windows or Linux and it has to be targeted to 32 or 64 bit architectures. This may be handled by an advanced build process which contains a cross-compiler that creates a compiled binary for each targeted platform. Additionally, a mechanism that chooses and loads the native binary, which is suitable for the current system architecture, has to be added to the implementation. This is achieved by adjusting the native wrapper class (cf. Figure 6.18). Nevertheless, if a new platform is to be supported, the build process as well as the native wrapper class have to be adjusted.

Another issue affects the state of natively implemented components. Atomic MontiArc components can be instantiated multiple times as subcomponents. Each instantiated subcomponent is represented by a distinct object in the simulation. According to Chapter 3, MontiArc components encapsulate their internal realization and do not share their state. If synchronisation of the

[8]The GNU Compiler Collection, `http://gcc.gnu.org/`.

state of distinct components is needed, e.g., to implement protocol components, this has to be realized by explicitly exchanging messages. Therefore, component state has to be implemented by using non-static internal fields since static variables have the same value for all instances of a class.

This has to be considered for native implementations, too. It especially restricts the usage of imperative non-object-oriented languages such as C to implement component behaviour because stateful methods in such languages are realized using global variables. Java JNI statically loads a native implementation once. Thus, the values of global variables are shared between all Java objects which access such a stateful native implementation. In this way, all instances of an atomic component with a C implementation implicitly share their state over the native layer. For this reason, native components should be either stateless or should be explicitly marked to have side-effects. Such components with side-effects should have singleton character and, thus, be only instantiated once as subcomponent. However, components implemented in an object-oriented native language like C++ are not influenced by this problem. Here, an individual pointer to a dedicated native object can be stored on the Java side for each component instance. The pointer can then be used to access the methods of the corresponding native object.

6.4 Validation of MontiArc Models

Testing is essential to ensure that developed components act as expected. In this section we distinguish and describe two different test methods. First, techniques for model-based black-box testing are described. Black-box tests are derived from the specification and the requirements of a component to validate its interface behavior. They can be used to define unit tests for atomic components as well as integration tests for decomposed components. Second, suitable techniques for white-box testing are presented. White-box tests are derived from the internal composition of a component to validate the expected flow of messages within a decomposed system. They are most suitable to test scenarios of interacting subcomponents. Consequently, they are used to complement black-box integration tests and to define system tests especially for closed systems that cannot be tested with black-box tests.

6.4.1 Model-Based Black-box Tests

A black-box or behavioral test is a test method which checks the functionality of a system without knowledge about the concrete implementation. Black-box tests are derived from an external view of the system under test, e.g., system requirements (see [Bin00]). This way, the external observable behavior is tested and not the internal structure of a system. Since MontiArc components interact via their external interface defined by ports, this test method is most suitable to validate component behavior. A black-box component test then has to compare the produced output of a component, which has been stimulated with a certain input sequence, with the expected output. This input-output relation has to be derived from the components requirements and should be intuitively describable in a test case.

Test Language Concepts*

MontiArc's communication semantics is based on streams which allow to describe an event based, timed communication. For this reason, it is most intuitive to define the aforementioned input sequences and the expected output in a notation similar to streams. The MontiArc framework provides a test DSL called *I/O-Test Language* that is inspired by test frameworks like JUnit [www13c]. The MontiCore grammar, that defines the language, is given in Listing C.3 on page 305. The concepts of the language are described in the following.

The metamodel of the I/O-Test language, which is depicted in Figure 6.21, gives an overview over the structure of the test language and corresponds to its abstract syntax. An `ArcTestSuite` contains tests for a testee given by the `ComponentUnderTest`. The testee corresponds to a component instance. Hence, configuration parameters and generic type parameters of components have to be set. In addition, a test suite contains elements of type `TestSuiteElement`. It is extended by interface `FieldDeclaration` which is implemented by concrete classes that realize *field variable* declarations, *stream field* declarations, and *stream matcher* declarations (not shown in the figure). Field variables can be used as values in an input or expected stream. Stream fields declare a complete stream definition and assign it to a variable which can be used as test input. In contrast, stream matchers declare (optionally underspecified) streams which can be used as expected streams only. Declared fields and matchers are visible within the scope of the complete test suite.

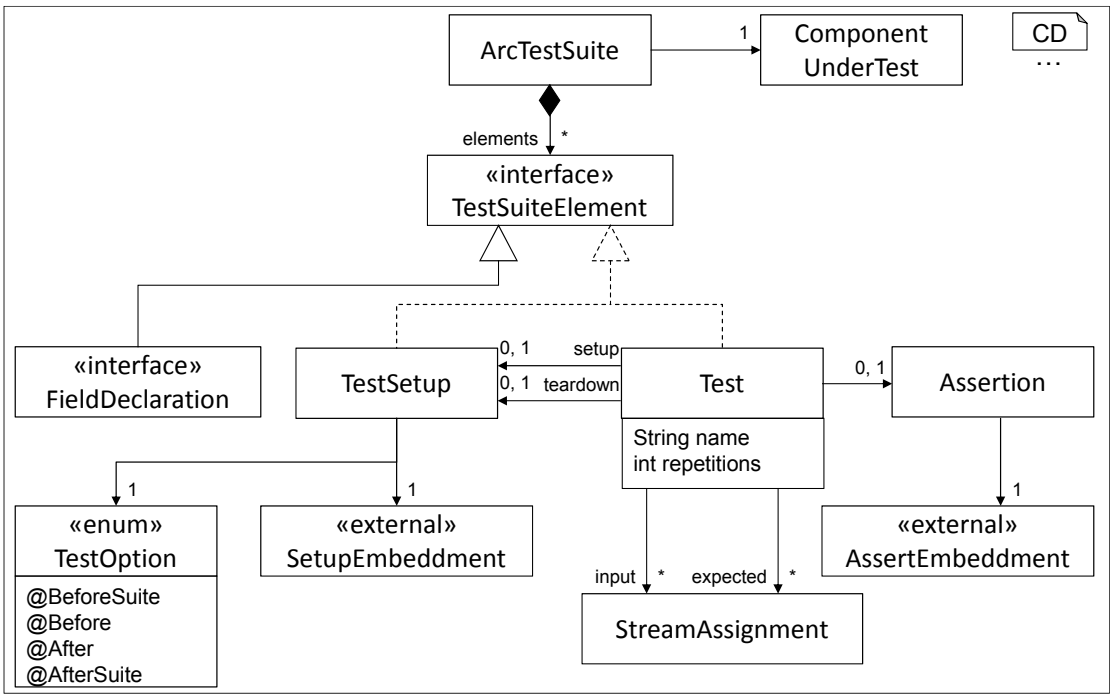


Figure 6.21: Metamodell with the most important elements of the I/O-Test Language.

A `TestSetup` is used to define additional setup or tear down operations. It has a `TestOp-`

tion that determines when the TestSetup has to be executed. Similar to JUnit, TestSetups marked with @BeforeSuite will be executed before the complete suite is started. Setups annotated with @Before respectively @After are executed before respectively after each test. @AfterSuite setups are executed after the complete suite has finished. The test language for MontiArc embeds Java statements to implement TestSetups.

A Test always has a name and can be repeated for a certain amount of times (repetitions). It optionally has a local setup or tear down which is executed for this test only. It can have an Assertion block which is used to define further assertions. Again, the test language for MontiArc embeds Java statements for this purpose. The input sequences of a test are given by input StreamAssignments. These are uniquely identified by their name which has to correspond to an incoming port of the testee. Expected results are given by expected StreamAssignments for each outgoing port of the testee. Consequently, the name of an expected stream assignment has to match the name of the corresponding outgoing port.

Creating Tests

The default configuration of MontiArc projects processes test models located in directory src/test/models. Further, test models are organized in packages. By convention, a test model should be located in the package that corresponds to the package of the component under test. To create a new test model, right-click the package in which the test should be created and select "New" and "Other ...". In the appearing wizard select "MontiArc-IO-Test" and press "Next". Enter the file name ABPSenderTest and select the testee component ABPSender after pressing the "Define component under test" button. Finally, press "Finish" to create an empty testsuite model for the testee.

To create a concrete test press Ctrl + Space and select "Test - Insert a new test case" from the autocompletion proposals. The editor will then insert a test template with an input and expect block which already contain StreamAssignments for each incoming respectively outgoing port of the testee. Since we do have to handle and check some non-primitive messages that are emitted by port abpMessage, we first setup some variables which are later used for this purpose (see Listing 6.22). Variables msg1 and msg2 are declared and values are assigned in ll. 1f. Variables abpMsg1 and abpMsg2 are declared in ll. 3f. A value is assigned to them in the @Before block (cf. ll. 6-9) which is executed before each test case of the testsuite. In this way, an ABPMessage with the payload msg1 and the acknowledgement flag true is assigned to variable abpMsg1. Variable abpMsg2 holds an ABPMessage with msg2 as its payload and an opposing acknowledgement flag.

Using these variables, we now test requirement *ABP_S5.1*. It claims that accepted data messages are encapsulated into ABPMessages and emitted via port abpMessage. A suitable test case is depicted in Listing 6.23. Message msg is accepted by port message (cf. l. 3), no input is defined for port ack (cf. l.4). In the same time interval, we expect message abpMsg1 via port abpMessage as the computation result (cf. l.7).

As claimed by requirement *ABP_S6.4*, messages have to be resent after a timeout of three time intervals. In FOCUS, progress of time is modeled using the notation of a tick ($\surd$, see Section 4.1). In the test language, a $\surd$ is represented by the value Tk. To ease modeling, values and value groups can be repeated in a stream. We utilize this by defining a test for

```
String msg1 = "Hello";                                          I/O-Test
String msg2 = "MontiArc";                                         ...
ABPMessage abpMsg1;
ABPMessage abpMsg2;

@Before {
  abpMsg1 = new ABPMessage(true, msg1);
  abpMsg2 = new ABPMessage(false, msg2);
}
```

Listing 6.22: Setup `ABPMessage` message objects to be used in the `ABPSender` test.

```
test encapsulateMsg {                                           I/O-Test
  input {                                                         ...
    message : <msg1>;
    ack : <>;
  }
  expect {
    abpMessage : <abpMsg1>;
  }
}
```

Listing 6.23: Test proper message encapsulation of component `ABPSender`.

the aforementioned requirement. The test input is given by message `msg1` on port `message` followed by 15 empty time intervals (15 * Tk) and 15 empty time intervals on port `ack`. The expected result for port `abpMessage` corresponds to $15/3 = 5$ times message `abpMsg1` each followed by three empty time intervals. Since the sender emits buffered messages at the start of a time interval, we additionally expect another `abpMsg1` at the end of the result stream. The resulting test case is depicted in Listing 6.24. More tests are depicted in Section E.2 on page 320. A description of further test concepts like optional messages, negated messages, or ranges can be found on the MontiArc website[9].

```
test repeatMessage {                                            I/O-Test
  input {                                                         ...
    message : <msg1, 15 * Tk>;
    ack     : <15 * Tk>;
  }
  expect {
    abpMessage : <5 * (abpMsg1, 3 * Tk), abpMsg1>;
  }
}
```

Listing 6.24: Timeout test for the `ABPSender`.

[9] How-To: Testing MontiArc Components, `http://www.monticore.de/languages/montiarc/howtotest/`.

The execution of I/O-Test models is directly integrated into the Maven build of a MontiArc project. Therefore, running `Maven install` on the project will automatically generate unit tests from the defined test models and executes them afterwards. If all tests succeed, the build will execute successfully. If tests fail, the build will fail as well and unexpected messages as well as current output traces are logged and printed to the console (see Figure 6.25).

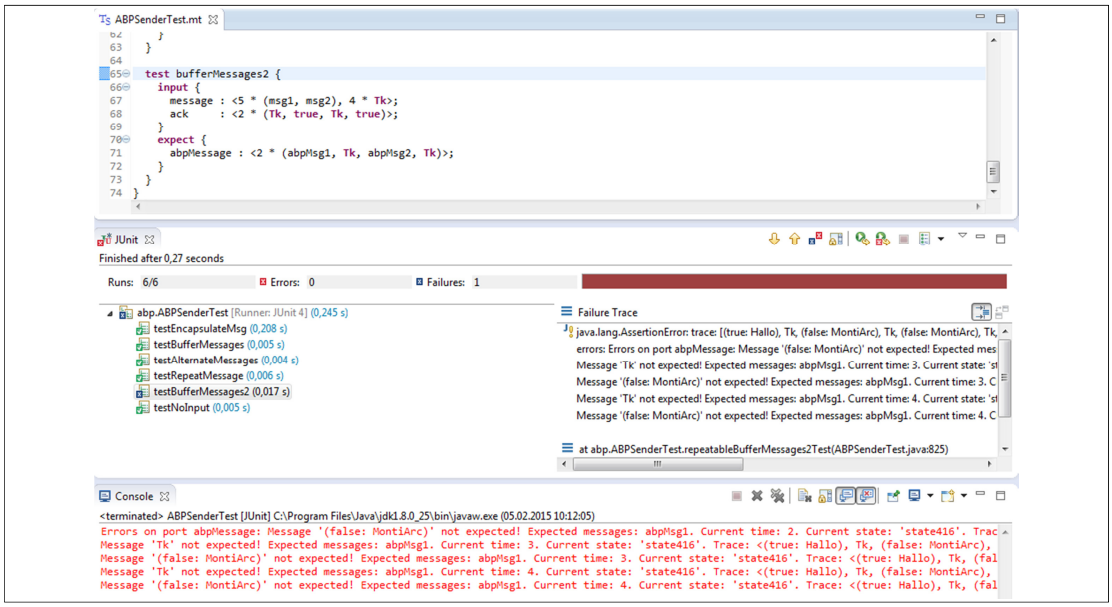


Figure 6.25: Screenshot of executed I/O-Tests with the failing test `bufferMessages2`.

6.4.2 White-box Testing of Decomposed Models

To test expected message flows in a decomposed system, the aforementioned black-box tests are not suitable since they test interface behavior of components only. Therefore, white-box or implementation-based test techniques are needed that are described in the following. They rely on an analysis of the internal structure respectively decomposition of the component under test (see [Bin00]). To validate message flows in a deterministic way, three different white-box test techniques, that consecutively build upon each other, are presented in the following.

- Message flows can be already validated by a minimal-invasive port instrumentation.
- Parameter manipulations are used to influence subcomponent instantiation and parametrization.
- Finally, complete subcomponents of the component under test can be replaced with mocks that mimic their behavior in a controlled way.

All three test techniques are realized using the JUnit [www13c] framework and are consequently implemented as JUnit test classes. In contrast to black-box tests, white-box tests are not defined using a dedicated modeling language. Since these tests require the flexible combination of test and setup techniques that also have to be part of a test model, a generative approach for white-

box tests is not worthwhile.

Testing Message Flows

To validate the flows between the subcomponents of our `ABPSender` component, we now create a white-box test. In this tutorial, we want to test that the transmission timer correctly notifies the inner sender after it has been activated by the latter. Therefore, we want to observe the highlighted connectors depicted in Figure 6.26.

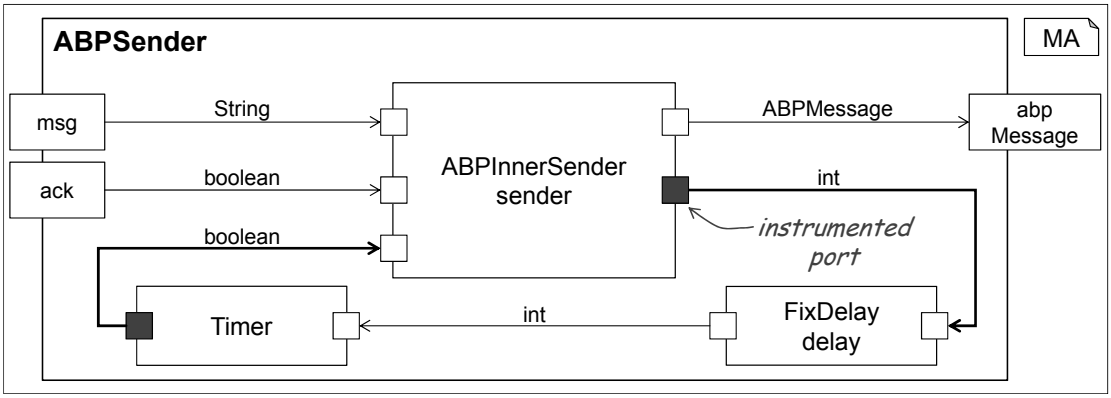


Figure 6.26: Instrumented ports of ABPSender's subcomponents in a white-box test.

At first, we create a test class in package `abp.gen` in the designated Java test class directory `src/test/java`. In this class, the testee of decomposed component `ABPSender` has to be setup correctly. Therefore, we create a field `testee` with the generated component class as its type. Please note, for black-box tests the generated component interface is sufficient as type since only access to ports is needed. A white-box test additionally needs access to the contained subcomponents. To get access to these subcomponents via the protected corresponding getters, the white-box test class and the generated component class have to to be in the same package.

A suitable test setup for a white-box test, which is able to test message flows, is depicted in Listing 6.27. First, we create an `ABPSender` instance with which we initialize the testee field (cf. l. 7). Then, we have to setup the testee with a scheduler that uses a `TestPortFactory` to produce ports and an error handler (cf. ll. 8-10). A `TestPort` is a special port implementation that stores messages which are transmitted over this port in a timed stream. The stored messages can be later on analysed in a test case. To observe the highlighted connectors in the figure, we simply cast the sending ports to `ITestPort` and acquire the streams which are transmitted over these ports (cf. ll. 12-15). For example, we get subcomponent sender from the testee (`testee.getSender`), get its outgoing port `setTimer` by calling `getSetTimer()`, cast it to `ITestPort` and then get the reference to the transmitted stream (`getStream()`). Please note that casting these ports to a test port is possible since we configured the scheduler to use a test port factory for port creation (cf. l. 9).

Subsequently, we are able to create concrete tests with our instrumented testee. Listing 6.28 contains a suitable test method which validates the aforementioned described message flow. The

JUnit

```
1 private ABPSender testee;
2 private IStream<Integer> setTimerStream;
3 private IStream<Boolean> timerEventStream;
4
5 @Before
6 public void setUp() {
7   testee = new ABPSender();
8   IScheduler s = SchedulerFactory.createDefaultScheduler();
9   s.setPortFactory(new TestPortFactory());
10  testee.setup(s, new SimpleErrorHandler());
11
12  setTimerStream = ((ITestPort<Integer>)
13      testee.getSender().getSetTimer()).getStream();
14  timerEventStream = ((ITestPort<Boolean>)
15      testee.getTimer().getTrigger()).getStream();
16 }
```

Listing 6.27: Setup method of a white-box test for component `ABPSender`.

concrete test can be divided into the following steps:

1. Sending a message and ticks using the testee. For this, we get port `message` from the testee and call its `accept` method with the message to transmit (cf. l. 8). Please note that method `tick` sends a $\surd$ to both incoming ports of the testee.
2. The produced streams of the instrumented ports are analyzed in a for-loop (cf. ll. 15 – 35).
3. First, we check that the timer is activated by the sender in the time intervals 1, 4, and 7 (cf. ll. 17 – 23).
4. Second, the expected reaction of the timer is checked in time intervals 3, 6, and 9 (cf. ll. 25 – 29).
5. Finally, it is validated that no other messages are transmitted in the other time intervals (cf. ll. 31 – 34).

This test setup allows to test message flows within decomposed, deterministic components since the input messages needed to generate the expected message flows can be determined before the test is executed. This is not possible with non-deterministic components which, e.g., contain lossy channels with random messages loss. In this case, it is not possible to determine the exactly needed number of input messages. For this reason, it is more suitable to instrument the testee in a way that allows us to avoid non-determinism. If the random behavior of a component is controlled by component parameters, we are able to disable non-determinism using a simple parameter manipulation.

Parameter Manipulation

Parameter manipulation allows to manipulate values assigned to component parameters during the instantiation of a parameterizable component without manipulating the implementation of the component. It is a suitable method to exchange the configured `IRandomFunction` of the

JUnit

```
1 @Test
2 public void testTimeOutTrigger() {
3   String msg = "Hello MontiArc";
4   int expectedTimeOut = 3;
5
6   // initial tick and message
7   tick();
8   testee.getMessage().accept(msg);
9
10  // send more ticks
11  for (int i = 0; i < 3 * expectedTimeOut; i++) {
12    tick();
13  }
14  // analyze the resulting streams
15  for (int i = 0; i < 3 * expectedTimeOut; i++) {
16    // check messages of port setTimer
17    if (i % expectedTimeOut == 1) {
18      assertFalse(setTimerStream.getTimeInterval(i).isEmpty());
19      assertEquals(Integer.valueOf(expectedTimeOut - 1),
20          setTimerStream.getTimeInterval(i).get(0));
21
22      assertTrue(timerEventStream.getTimeInterval(i).isEmpty());
23    }
24    // check expected timeout
25    else if (i % expectedTimeOut == 0 && i > 0) {
26      assertTrue(setTimerStream.getTimeInterval(i).isEmpty());
27      assertFalse(timerEventStream.getTimeInterval(i).isEmpty());
28      assertTrue(timerEventStream.getTimeInterval(i).get(0));
29    }
30    // no messages in other time intervals
31    else {
32      assertTrue(setTimerStream.getTimeInterval(i).isEmpty());
33      assertTrue(timerEventStream.getTimeInterval(i).isEmpty());
34    }
35  }
36 }
```

Listing 6.28: Implementation of a white-box test for component `ABPSender`.

library component `LossyDelayedChannel` to adjust message loss for a specific test case.

Subcomponent objects in the MontiArc simulation are created by factories which offer a method to register concrete factories to be used for object creation (see Section 5.5.2). Utilizing this, we now create a local test factory as an inner class of our test class which extends the factory for component `LossyDelayedChannel`. As depicted in Listing 6.29, it overwrites the dynamic object creating method `doCreate` from the superclass. The overwriting method simply delegates to the overwritten method but replaces the given `IRandomFunction f` with a `ControllerRandom` which always returns `true` (cf. l. 7). In this way, the test factory solely

produces delayed channels which do not loose messages.

```
1 class LossyDelayedChannelTestFactory extends                    JUnit
2     LossyDelayedChannelFactory {
3   @Override
4   protected <T> ILossyDelayedChannel<T> doCreate(
5       IRandomFunction f, int delay) {
6     // always set loss rate to zero
7     return super.doCreate(new ControlledRandom("1"), delay);
8   }
9 }
```

Listing 6.29: A local test factory to manipulate parameter values of `LossyDelayedChannel` subcomponents.

We continue to adjust the test setup by registering a new instance of the test factory at the `LossyDelayedChannelFactory` as shown in Listing 6.30. After the test is executed, we reset the behavior of the default factory by calling its `reset()` method (cf. ll. 6-8). The complete white-box test for component `ABP` which uses parameter manipulation is depicted in Listing E.7 on page 324.

```
1 private void setUpFactory() {                                   JUnit
2   // register test factory
3   LossyDelayedChannelFactory.register(
4       new LossyDelayedChannelTestFactory());
5 }
6
7 @After
8 public void tearDown() {
9   LossyDelayedChannelFactory.reset();
10 }
```

Listing 6.30: Register the parameter manipulating local factory and clean up after test execution.

Mocking Components

In object oriented programs, a mock object is an object which mimics the behavior of another object in a controlled way. According to Freeman and Pryce [FP09], it corresponds to a substitute implementation which is used to test how an object interacts with related objects. If the behavior of a more complex decomposed MontiArc model is to be tested, not all subcomponents are of interest. If we, for example, only want to examine the interaction between sender and receiver in the ABP component, we simply can mock the lossy delayed channels. Since we are able to completely exchange the implementation of subcomponents of the testee, mocking is more powerful than parameter manipulation.

To create a mock for the subcomponents of type `LossyDelayedChannel`, we create a local class which implements the mock's behavior. If an atomic component is to be mocked, we can simply choose the generated abstract class as superclass. If we want to mock a decomposed component, we can use a stub class as superclass which can be also generated by MontiArc[10]. For decomposed component type `LossyDelayedChannel` we use class `ALossyDelayedChannelStub` as depicted in Listing 6.31. We then can implement the behavior of the mock as explained in Section 6.3. The given example simply forwards messages received on port `portIn` to port `portOut`. Additionally, the component counts how many messages have been transmitted and stores the last transmitted message. To replace the default generated implementation of component `LossyDelayedChannel` with the mock implementation, we have to create and register a dedicated mock factory as described in the previous section. The complete test class is depicted in Listing E.8 on page 326.

JUnit

```
1 class LDCMock<T> extends ALossyDelayedChannelStub<T> {
2   public int called = 0;
3   public T lastMessage;
4
5   public LDCMock(IRandomFunction f, int delay) {
6     super(f, delay);
7   }
8
9   @Override
10  protected void treatPortIn(T message) {
11    sendPortOut(message);
12    called++;
13    lastMessage = message;
14  }
15 }
```

Listing 6.31: Mock implementation of decomposed component `LossyDelayedChannel`.

How to access and use the mocked subcomponents is depicted in Listing 6.32. At first, both subcomponents are acquired from the testee (cf. ll. 2f). Please note that these need to be casted to the concrete mock class `LDCMock` since the return type of the testee's methods `getMed1()` and `getMed2()` is `ILossyDelayedChannel`. Then 1000 messages are transmitted in a for-loop and for each message it is checked, whether the expected message (cf. l. 10) and the alternating acknowledgement (cf. ll. 12f, 16f) are transmitted.

6.5 Generalize Components

MontiArc provides *generic component types* which allow to define generic type parameters that can be used as port and configuration parameter data types within the component type definition. These generic types have to be assigned when instantiating a generic component type as

[10] Configure the MontiArc Maven Plugin with parameter `generateStubs`; cf. Section 5.7.2 on page 176.

JUnit

```
1 public void testAlternatingBit() {
2   LDCMock<ABPMessage> m1 = (LDCMock<ABPMessage>)
3       testee.getMed1();
4   LDCMock<Boolean> m2 = (LDCMock<Boolean>) testee.getMed2();
5   int amount = 1000;
6   for (int i = 0; i < amount; i++) {
7     String msg = "Msg " + i;
8     testee.getMsg().accept(msg);
9     assertEquals(i + 1, m1.called);
10    assertEquals(i + 1, m2.called);
11    assertEquals(msg, m1.lastMessage.getContent());
12    if (i % 2 == 0) {
13      assertTrue(m1.lastMessage.isAck());
14      assertTrue(m2.lastMessage);
15    }
16    else {
17      assertFalse(m1.lastMessage.isAck());
18      assertFalse(m2.lastMessage);
19    }
20  }
21  assertEquals(amount, receiverOut.getStream().size());
22 }
```

Listing 6.32: Using mocked subcomponents in a test case.

subcomponent. This concept is useful for components with a behavior that does not depend on specific data of an accepted message but only on the event of receiving data.

The ABP component and its subcomponents developed in this tutorial do not depend on the actually transmitted payload but react to the event of a message which is to be transmitted. Therefore, it is well suited to be realized as a generic component. Additionally, several parameters that influence the behavior of the protocol are hard coded in the model or the behavior implementation of atomic components. These are the transmission delay, the function that controls the message loss of the lossy delayed channels, and the retransmission timeout. These parameters are parameterizable as well if component parameters are added to the component type definitions. In the following steps the ABP is generalized by applying these techniques.

Adjust container data type: First, we adjust the `ABPMessage` container data type which is used to encapsulate data messages together with acknowledgment flags. Thus, we add a generic type parameter `T` to the class definition and replace the former type `String` of the field `content` with `T`.

Adjust component sender: The modifications of the `ABPSender` component are contained in Listing 6.33 and stepwise listed in the following:

1. We adjust the `ABPSender` model by adding a generic type `T` to the component definition as well as a parameter `timeout` with type `int` (cf. l. 5).
2. We change the type of port `message` to `T` and the type of port `abpMessage` to `ABPMessage<T>` (cf. ll. 9, 11).

3. We repeat these modifications for the inner component definition `ABPInnerSender` (cf. ll. 13, 15, 17).
4. Automatic instantiation of inner components cannot be applied to generic and/or parameterizable components. Thus, an `ABPInnerSender` subcomponent is added (cf. l. 22).

MA

```
1 package abp;
2 import ma.sim.FixDelay;
3 import ma.util.Timer;
4
5 component ABPSender<T>[int timeout] {
6   autoconnect port;
7
8   port
9     in T message,
10    in Boolean ack,
11    out ABPMessage<T> abpMessage;
12
13  component ABPInnerSender<T>[int t] {
14    port
15      in T message,
16      in Boolean ack,
17      out ABPMessage<T> abpMessage;
18    port
19      out Integer setTimer,
20      in Boolean timerEvent;
21  }
22  component ABPInnerSender<T>(timeout) sender;
23  component FixDelay<Boolean>(1) delay;
24  component Timer;
25
26  connect timer.trigger -> delay.portIn;
27  connect delay.portOut -> sender.timerEvent;
28 }
```

Listing 6.33: Generalized component `ABPSender`.

Adjust component receiver: The component definition `ABPReceiver` is adjusted in a similar way by adding a generic type parameter `T`. We then use `T` as port type for outgoing port `message` and use it to parametrize type `ABPMessage` from port `abpMessage`. The resulting component model is depicted in Listing E.9 on page 328.

Adjust component ABP: The `ABP` component is generalized with the following steps:

1. Add generic type parameter `T` to component `ABP`.
2. Add configuration parameters:
   a) `IRandomFunction rand` to control message loss of the lossy delayed channels,
   b) `int delay` to configure the delay of the contained lossy delayed channels, and
   c) `int timeout` to configure the message resent timeout of the contained sender.

3. Replace the port data types `String` with `T`.
4. Parametrize the contained subcomponents to fix the errors which report an invalid amount of type parameters. For this purpose:
   a) add parameter `T` to the type `ABPSender` and `ABPReceiver` of subcomponents `sender` and `receiver` and
   b) adjust the type parameters of subcomponent `med1` to use `ABPMessage<T>`.
5. Replace the hard coded `new ControlledRandom(...)` value object which has been used for `med1` and `med2` with the parameter `rand`.
6. Replace the hard coded delay of these subcomponents with parameter `delay` as well.
7. Configure subcomponent `sender` with parameter `timeout` to fix the remaining error that reports an invalid amount of configuration parameters.

The resulting component model is depicted in Listing E.10 on page 328.

Adjust build configuration: Open the build configuration (`pom.xml`) and navigate to the configuration of the `montiarc-maven-plugin`. Switch parameter `generateInteractiveSimulation` to false and save. Then execute `Maven clean install`. Since the implementation of the atomic components is not adjusted yet, the build will fail. However, the resulting compiler errors give good hints, which manually implemented classes need to be adjusted in the following steps.

Adjust behavior implementations: Implementations are adjusted with the following steps:

1. Add generic type `<T>` to the behavior implementations `ABPInnerSenderImpl` and `ABPReceiverImpl` created in Section 6.3 and pass this type to the extended superclass.
2. Method `treatAbpMessage` of class `ABPReceiverImpl` has to handle parameters with type `ABPMessage<T>` now.
3. Adjust class ABPInnerSenderImpl:
   a) The `buffer` now has to handle messages of type `T` instead of `String`.
   b) Methods `treatMessage` and `sendMessage` have to handle messages with type `T` instead of `String`.
   c) Thus, the latter method has to create and emit messages with type `ABPMessage<T>`.
   d) Since component `ABPInnerSender` is parameterizable with a timeout parameter now, the constructor of the corresponding behavior implementation requires this parameter, too. Therefore, add parameter `int timeout` to the constructor and pass it to the constructor of the superclass.

Adjust test implementations: Finally, we have to adjust the test models created in Section 6.4. The test models are changed by parameterizing the testee with the previously used message type `String` and the random function `new ControlledRandom("1010")`, 1 (one-way delay) and 3 (timeout). Of course the types of defined fields have to be changed from `ABPMessage` to `ABPMessage<String>` as well. This is demonstrated by means of the test `ABPSenderTest` which is depicted in Listing 6.34.

Finally, the handwritten tests have to be adjusted. Therefore, every instantiated component class has to be parametrized correctly with `<String>` and the aforementioned parameter set is to be used in constructor and factory calls. Every unparametrized type `ABPMessage` has to be replaced with a parametrized version `ABPMessage<String>`. Rebuild the project by

executing `Maven clean install`. If everything has been adjusted correctly, the project builds successfully.

```
1 testsuite ABPSenderTest for ABPSender<String>(3) {                I/O-Test
2   String msg = "Hallo";
3   String msg2 = "MontiArc";
4   ABPMessage<String> abpMsg1;
5   ABPMessage<String> abpMsg2;
6   @Before {
7     abpMsg1 = new ABPMessage<String>(true, msg);
8     abpMsg2 = new ABPMessage<String>(false, msg2);
9   }
```

Listing 6.34: Adjusted `ABPSender` I/O-Test.

6.6 Optimization Testing

Parameterizable components can be easily statistically analysed by iterative experiments with distinct parameter configurations. Known parameters are fixed and parameters that shall be optimized are varied in iterative tests with identical input messages. The produced results are stored and compared regarding the property that is to be optimized, e.g., passed total simulation time or average transmission time. In contrast to regular component validation tests (see Section 6.4), a strict deterministic behavior of all involved components is not mandatory.

How to set up an optimization test for the ABP is depicted in Listing 6.35. Please note that the complete implementation is given in Listing E.11 on page 329. It simulates 5000 single message transmissions with various combinations of loss rates between 10 and 80 percent and delays between 1 and 10 time intervals. It measures the complete round-trip time needed to deliver and acknowledge the message. Therefore, the testee is set up similar to a white-box test (cf. ll. 12 – 18), a single message is sent (cf. l. 21), and Ticks are sent until message transmission is acknowledged (cf. ll. 22 – 25). The passed simulation time is stored in a `DescriptiveStatistics` object (cf. ll. 26 – 28) to collect statistical information. Finally, the results are stored and organized in a spread sheet for further analyses (cf. ll. 33 – 35). Please note that this example uses the Apache Commons Math library [11] for statistical calculations and the Apache POI library[12] to export the spread sheet.

In total, this optimization test setup iteratively executes 400.000 simulations. Since these simulations are independent and do not influence each other, they are well suited to be executed in parallel. The complete implementation of a parallelized version is given in Listing E.12 on page 332. The involved classes are depicted in Figure 6.36. The `SimulationTask` is a `Runnable` which implements the execution of a single simulation. It is initialized with a `CountDownLatch` and a `SimResult`. The former is shared among all SimulationTasks and is notified when a simulation is finished by calling its `countDown` method. It is used to synchronize the main execution thread with the parallel simulation threads (see [Blo08, Item

[11] Commons Math: The Apache Commons Mathematics Library, `https://commons.apache.org/proper/commons-math/`.

[12] Apache POI - the Java API for Microsoft Documents, `https://poi.apache.org/`.

69]). The simulation results are submitted to the `SimResult` using its `addResult` method. In this way, the result is added to the associated `DescriptiveStatistics`.

A suitable way to execute `SimulationTasks` in parallel is depicted in Listing 6.37. The `ExecutorService` provides a thread pool which is restricted to the amount of available processors respectively cores (cf. ll. 1f). The used `CountDownLatch` is initialized to the total number of simulations, i.e., eight different loss rates combined with 10 different delays and each combination is repeated 5000 (`EXPERIMENT_AMOUNT`) times, which results in 400.000 simu-

```java
final int experimentAmount = 5000;
final Tick<String> tick = Tick.<String> get();
final String transmittedMsg = "Hello MontiArc";

for (int lossrate = 10; lossrate <= 80; lossrate += 10) {
  for (int delay = 1; delay <= 10; delay++) {
    int timeout = 2 * delay + 1;
    // from apache commons math
    final DescriptiveStatistics ds = new DescriptiveStatistics();
    for (int i = 0; i < experimentAmount; i++) {
      // setup ABP
      ABP<String> abp = new ABP<String>(new JavaRandom(lossrate),
          delay, timeout);
      IScheduler s = SchedulerFactory.createDefaultScheduler();
      s.setPortFactory(new TestPortFactory());
      abp.setup(s, new SimpleErrorHandler());
      TestPort<Boolean> ack = (TestPort<Boolean>)
          abp.getSender().getAck();

      // sent message
      abp.getMsg().accept(transmittedMsg);
      // sent ticks until msg is acknowledged
      while (ack.getStream().getUntimedHistory().isEmpty()) {
        abp.getMsg().accept(tick);
      }
      // passed simulation time
      int time = ack.getStream().getCurrentTime();
      ds.addValue(time);
    }
    System.out.println("Computed statistics for ABP(" + lossrate +
        ", " + delay + ", " + timeout + ")");
    rownum++;
    addToTable(sh, rownum, lossrate, delay, timeout, ds.getMin(),
        ds.getMax(), ds.getMean(), ds.getStandardDeviation(),
        ds.getPercentile(50));
  }
}
```

Listing 6.35: Setup of an optimization test for component ABP.

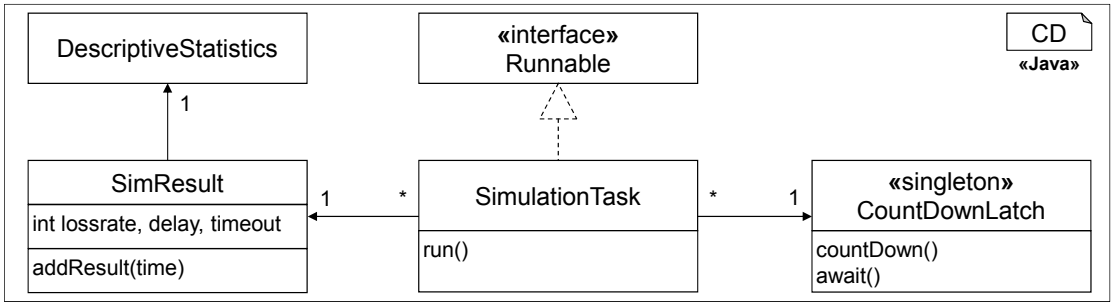


Figure 6.36: Involved classes of a parallelized optimization test.

lations. In the nested for-loops, a `SimResult` is created for each parameter configuration (cf. l. 13). It is passed with the latch to a `SimulationTask` in another nested for-loop (cf. ll. 15 – 17). The task is then submitted to the executor service. Finally, `latch.await()` (cf. l. 20) pauses the current thread to wait until all submitted tasks have finished. This is the case if the `countDown` method of the latch has been called 400.000 times. Afterwards, the results are exported to a spread sheet. On an Intel® Core™ i7 CPU Q 740 @ 1.73GHz, 64 Bit with 4 cores and 8 threads, the execution time is reduced from about 27 minutes to less than 6 minutes with the parallelized simulation. Thus, it is is about 4.5 times faster than the linear version.

Java

```
1 int cores = Runtime.getRuntime().availableProcessors();
2 ExecutorService exec = Executors.newFixedThreadPool(cores);
3 final int lossRateMax = 80;
4 final int delayMax = 10;
5 // used to monitor the amount of finished simulations
6 final CountDownLatch latch = new CountDownLatch(
7     (lossRateMax / 10) * delayMax * EXPERIMENT_AMOUNT);
8
9 List<SimResult> results = Lists.newLinkedList();
10 for (int lossr = 10; lossr <= lossRateMax; lossr += 10) {
11   for (int delay = 1; delay <= delayMax; delay++) {
12     int timeout = 2 * delay + 1;
13     SimResult r = new SimResult(lossr, delay, timeout);
14     results.add(r); // store result object
15     for (int exp = 0; exp < EXPERIMENT_AMOUNT; exp++) {
16       exec.submit(new SimulationTask(latch, r));
17     }
18   }
19 }
20 latch.await(); // wait until all tasks have finished
```

Listing 6.37: Executing a parallel optimization test.

6.7 Documentation of MontiArc Models

Documentation is important to support reuse of artifacts which encapsulate their internal realization. MontiArc components hide their internal implementation and provide their functionality via a defined public interface. In this way, decomposed and atomic components can be used the same way. The interface of a MontiArc component is defined by a set of incoming and outgoing ports which can receive or emit messages of a certain type. However, the concrete relation between incoming and outgoing messages is not given in the interface. Hence, which reaction is produced by a component for a certain stimulus is hidden in its internal realization. Thus, further documentation is needed to enable reuse of MontiArc components without detailed knowledge of their implementation respectively composition.

The MontiArc language contains comments which annotate components or elements of the component. These comments can be used to document the interface of a component as well as the resulting behavior for received input events. The MontiArc framework provides a documentation generator which has initially been developed in the bachelor theses [Hom12]. Based on the MontiArc component itself and the contained comments, it produces HTML documentation artifacts. Similar to Javadoc[13], a tool for generating application programming interface (API) documentations for Java classes, it has the advantage to keep the documentation next to the documented artifact. For example, if a type parameter is added to a component, only the comment above the component definition needs to be adjusted. In this way, outdated parts of the documentation can be identified more easily and no external documents need to be adjusted.

6.7.1 Enabling the Documentation Generator

To enable the documentation generator, execute the goal `doc` of the MontiArc Maven plugin. For this purpose, the `pom.xml` has to be adjusted by adding the goal to the list of goals to be executed as depicted in Listing 6.38 line 14.

This setup is already sufficient to generate documentation for components into directory `target/madoc`. Nevertheless, the default configuration can be adjusted using the following parameters:

- `javaDocUrls`: Adds further URLs to JavaDoc documentations. These URLs are used to resolve links to the documentation of data types annotated with an `@link` tag (see Table 6.40). Resolved links are automatically included into the MontiArc documentation. Defaults to `http://docs.oracle.com/javase/7/docs/api/`.
- `isPublic`: Generate documentation of publicly visible component parts (interface, configuration, and type parameters). If set to `false`, documentation for, e.g., subcomponents is included. A complete list of public and protected model elements is given in Table 5.11 on page 139. Defaults to `true` and consequently generates documentation for public model elements.
- `modelDirectory`: The MontiArc model directory. Defaults to `src/main/models`.

[13] Javadoc Tool, `http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html`.

pom.xml

```
<build>
  <!-- ... -->
  <plugins>
    <!-- ... -->
    <plugin>
      <groupId>de.monticore.lang.montiarc</groupId>
      <artifactId>montiarc-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>clean</goal>
            <goal>configure</goal>
            <goal>generate</goal>
            <goal>doc</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Listing 6.38: MontiArc Maven plugin configuration to generate documentation.

- `docOutputDirectory`: The output directory for the generated documentation. Defaults to `target/madoc`.

6.7.2 Document Components

MontiArc's syntax supports comments in Java style. Thus, single-line comments start with '//' and multi-line comments start with '/*' and end with '*/'. Comments that should be included in the documentation are multi-line comments that start with '/**' and end with '*/'. These comments are always assigned to the next model element. Similar to Java, special documentation elements are controlled with tags. It is distinguished between single-line and multi-line tags. The former are enclosed by curly brackets and allow multi-line comments to be bound to one tag. The latter have to be used at the beginning of a comment line and are restricted to a single line. Additionally, multi-line tags can be used anywhere within a comment and thus do not have to be located at the beginning of a line. Single-line tags interpreted by the MontiArc documentation generator are listed and explained in Table 6.39, available multi-line tags are given in Table 6.40.

Tag	Description
*@author[s] $name (',' $name)**	Adds an authors segment to the documentation which contains the given author or the list of given authors.

Table 6.39 continued on next page

Tag	Description
@*brief $text*	Creates an introductory text for the model element which is printed into overview lists. For example, the text defined in a component's *brief* tag is printed in the package view of the documentation.
@*date $date?*	Adds a date segment to the documentation. If *$date* is missing, the current date (generation time) is used. If *$date* is given, it has to be given in format `DD.MM.YYYY` or `YYYY-MM-DD`.
@*hint $text*	Adds a hint segment to the documentation which contains the given *$text*. Hints can be used to reference the reader to further related components or information.
@*param $par $text*	Adds the given *$text* as documentation of configuration parameter *$par*. This tag is used to describe the effect of the corresponding parameter. If the configuration parameter does not exist, an error is thrown. This tag can also be used as a multi-line tag.
@*rev $text*	Adds the given *$text* to the revision segment of the documentation. The printed revision number can be useful to reconstruct changes of a component within a revision control system (RCS).
@*sideEffects $text*	Adds a side effects warning box to the top of the documentation which contains the given *$text*. Use this tag to document side effects of dirty component, e.g., read or write access to files (see Section 5.5.3). The documentation of side effects is important since components with side effects can act unpredictable within a MontiArc simulation. Consequently, the user of such component has to be informed about the impact of a side effect. For longer descriptions, this tag can also be used as a multi-line tag.
@*since $text*	Adds a since segment to the documentation which contains the given *$text*. Use this tag to document since when a component is available, e.g., in a library. It can be also used to annotate ports which have been added to an already existing component. Usually, the given *$text* references the version number of the MontiArc project.
@*state $text*	Adds the given documentation as description of the state-space of a component. Use this tag to document different reaction on the same input event in the different states of a component. For longer descriptions, this tag can also be used as a multi-line tag.
@*stateless*	Marks a component as stateless and adds this information to the state segment of the documentation. Use this tag to indicate that a component always produces the same output for the same input.

Table 6.39 continued on next page

Tag	Description
@type $par $text	The given *$text* is used as documentation for generic type parameter *$par*. Use this tag to describe which model elements use this parameter. If this parameter does not exist, an error is emitted.
@version $text?	Creates a version segment in the documentation and adds the given *$text*. If the optional *$text* is not given, the generator adds the text which has been passed via Maven configuration parameter `docVersion` to the executed documentation tool. If this parameter is not explicitly set, the version number of the Maven project is used. In this way, the version number within the documentation is always aligned with the current version number of the project.

Table 6.39: Single-line tags of the MontiArc documentation generator.

Tag	Description
{*@code $text*}	Formats the given *$text* in a typewriter font. This tag can be used to highlight certain parts of the documentation, e.g., preconditions or accepted value ranges of incoming ports.
{*@escape $text*}	Escapes the given *$text*. Thus, further contained tags are not interpreted and HTML parts are escaped. In this way, the complete given text is printed in the documentation.
{*@link $name*}	Creates a link to the documentation of the model element named *$name*. This can be used to reference the documentation of another model element. For example, ports can be referenced within the documentation of the component type to document the relation between input und output ports. If the model element does not exist in the current model, an error is emitted.
{*@link $type*}	Creates a link to the documentation of the given *$type*. Component and data types can be linked. This tag can be used to, e.g., create a link to the API documentation of the Java class that implements the behavior of an atomic component or to reference a used constant class or enumeration. Please note that *$type* has to be fully qualified.
{*@link $type#method*}	Creates a link to the documentation of the *method* which is defined in the given *$type*. This tag can be used to reference certain helper methods used within the component. The *$type* has to be fully qualified and the method can be optionally qualified with a list of parameters, if needed.

Table 6.40: Multi-line tags of the MontiArc documentation generator.

6.7.3 Index Page Design

It is possible to add further descriptions to the generated index page. If, for example, the current project contains a library of related components, the index page should contain further information about the library. To influence the design of the generated index page, simply create an HyperText Markup Language (HTML) file named `documentation_main.html` in the root model directory. In the default configuration, this is directory `src/main/models`. Then use regular HTML syntax to design the index page. In this way, images and links can be added to the index page. Beside regular HTML, the following special documentation tags are interpreted within this file:

- *@title $name*: Sets the given *$name* as title of the whole generated documentation. Usually, this should be the complete name of the library or the project (`name` element in the `pom.xml`).
- *@brief $text*: The given *$text* is used as text for the generated link that lists all contained components of the project in the package view. Usually, the artifact id of the project should be used for this tag (`artifactId` element in the `pom.xml`).
- *@version $text?*: Adds a version segment to the index page. It is processed like the `@version` tag documented in Table 6.39.

An exemplary index page that demonstrates the effect of these tags is depicted in Figure 6.41. The upper-left frame contains a list of links to the documentation of the packages. The lower-left frame contains a list of links to the component documentations. The index page is displayed in the right frame.

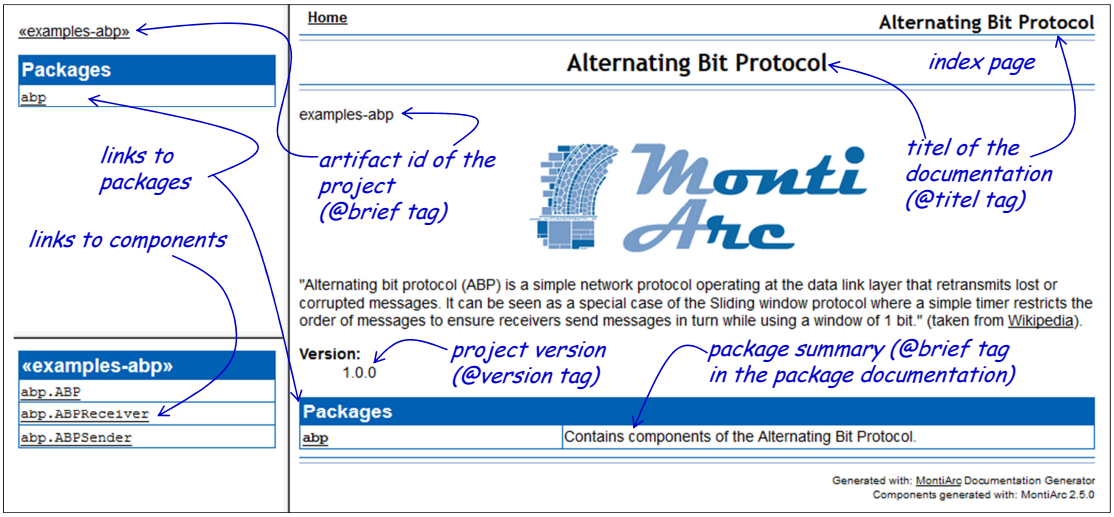


Figure 6.41: Index page of the generated ABP documentation.

6.7.4 Package Documentation

Similar to Java, a MontiArc package is represented by a directory and not by a file. Nevertheless, it is possible to create the documentation of a package to describe its meaning and the relation of the contained components. To document a package, create a HTML file named `package.html` in the directory which represents the package that is to be documented. Use regular HTML syntax to design the documentation. In contrast to the previously discussed index documentation, only the *@brief* tag is interpreted. The given text is displayed as a summary in the package overview of the generated documentation (see Figure 6.41).

6.8 MontiArc Libraries*

A library in a general purpose language (GPL) is a collection of behavior implementations which can be accessed and reused via well defined APIs. Following the principle of information hiding, the provided behavior can be reused without knowing the concrete implementation. Since MontiArc components encapsulate their internal realization and provide their behavior via well defined component interfaces, they are best suited to be bundled in reusable component libraries. Thus, a MontiArc library is a collection of reusable (logically related) components and their implementations which can be instantiated as subcomponents in other MontiArc components. To support the modeler and ease reuse of components, also the generated documentation is contained in MontiArc libraries.

6.8.1 Structure of a Model Library

MontiArc libraries are structured in three layers: the model layer, the implementation layer, and the documentation layer (see Figure 6.42). These layers contain distinct artifact kinds which are bundled in distinct library jar files.

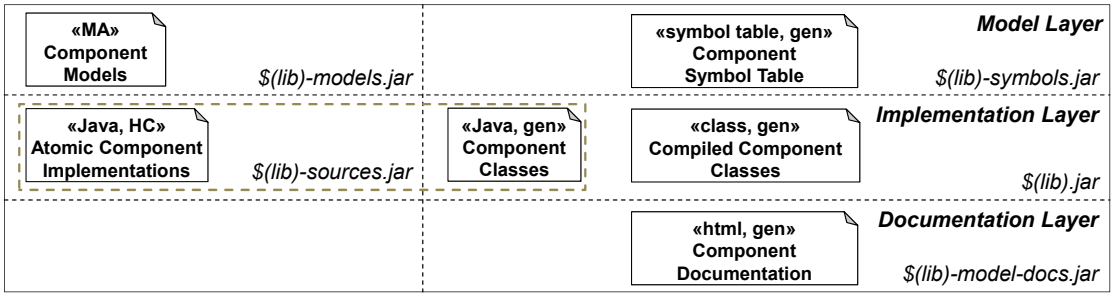


Figure 6.42: Layers of a component library.

The **Model Layer** contains handwritten MontiArc models and their corresponding serialized symbol table entries (see Section 5.2). The former are packaged in a jar file with the classifier `models`, while the latter are packaged in a jar file with the classifier `symbols`. At least one of these artifacts needs to be referenced in the modelpath (see Section 5.7) if a component from this library should be instantiated as a subcomponent.

The **Implementation Layer** contains simulation specific Java classes in source and binary form. The former are handwritten implementations of atomic components packaged together with the generated component classes into a jar file with classifier `sources`. The main jar file without classifier contains the compiled binaries of these files. The MontiArc generator generates an interface $\texttt{IT}_{gen}$ for each component type `T`. If `T` is instantiated as a subcomponent within a decomposed component `DC`, the generator produces a field for the subcomponent in the class $\texttt{DCC}_{gen}$ that is generated from component `DC`. This field has the type $\texttt{IT}_{gen}$. Consequently, a compile-time and runtime dependency between the classes $\texttt{DCC}_{gen}$ and $\texttt{IT}_{gen}$ exists. If component `T` is located in a library, the corresponding main jar file has to be added to the Java classpath to fulfill this compile-time and runtime dependency.

The **Documentation Layer** contains the HTML documentation of the contained components packaged in a jar file with classifier `model-docs`. The documentation is derived from comments given in the component models (see Section 6.7.2). To enable the MontiArc Eclipse IDE to resolve the contained documentation, this jar file has to be added to the Java classpath. In this way, information about library components is depicted in the autocompletion dialog or when the mouse is hovering over a component type.

The artifacts of the upper two layers are divided into human readable artifacts and generated binary artifacts. The former allow to reconstruct the internal implementation of components, i.e., handwritten sources. The latter are derived from the former category by processing models with the MontiArc tooling and implementations with the Java compiler. MontiArc only needs symbol table entries to load referenced components and the Java compiler only needs the compiled classes to fulfill the aforementioned compile-time and runtime dependencies. Consequently, a minimal library only consists of these binary jars. Consequently, this strict separation allows to release closed source libraries which neither contain MontiArc models nor the source code of atomic component implementations (see requirement *LRQ5.3*). Such a library only consists of the jar files located on the right-hand side of Figure 6.42.

6.8.2 Predefined Libraries

MontiArc provides a set of predefined libraries which can be reused within new projects. Projects created with the MontiArc project wizard (cf. Section 5.7.3) are already configured to reuse these libraries. Documentation of the contained library components can be found at the MontiArc website[14]. The following libraries are given:

- montiarc-lib-dt: The **MontiArc Digital Techniques Library** contains basic components for boolean calculations. Additionally some digital techniques components, e.g., Flip Flops, are given.
- montiarc-lib-img: The **MontiArc Image Library** contains components which allow to process images in a pipes-and-filters manner.
- montiarc-lib-math: The **MontiArc Math Library** provides basic components for mathematical calculations.

[14]MontiArc Component Library Documentation, `http://www.monticore.de/languages/montiarc/lib/`.

- montiarc-lib-monitor: The **MontiArc Monitoring Library** provides components to monitor systems and services. It has been developed in a bachelor thesis [Ix12]. It contains agent components used to monitor system properties (e.g., CPU or RAM usage) and monitor components which observe external services. These components can be connected to controller components which collect reported incidents, generate reports, and transmits them using components from the Network Library.
- montiarc-lib-net: The **MontiArc Network Library** contains components which provide access to Internet based services or protocols. Among these are e-mail and instant messaging components.
- montiarc-lib-sim: The **MontiArc Simulation Library** contains simulation specific components used to simulate transmission delays or loss of messages. It further contains sink and source components to display and inject messages in decomposed components.
- montiarc-lib-util: The **MontiArc Utility Library** provides a collection of utility components. It comprises components to route signals (merge, split), to access and process files (text, serialized data, compressed archives), and simple encryption and conversion components.

6.8.3 Creating a Library

Actually, all MontiArc projects that at least produce and publish the binary artifacts described in Section 6.8.1 can be reused as a library. However, this is not very convenient since at least two dependencies have to be added to the project that wants to reuse a library. The `models` or `symbols` jar has to be added to the modelpath and the main binary jar has to be added to the classpath. Optionally, to increase the usability of a library, the `model-docs` jar has to be added to the classpath, too.

To ease library reuse, the exported artifacts of a library should be bundled in an aggregating Maven project. For a detailed description of Maven we refer to [www14k]. An excerpt of an exemplary aggregating project configuration (`pom.xml`) is depicted in Listing 6.43. As a convention, the group id (cf. l. 1) corresponds to the group id of the aggregated library (cf. ll. 7, 12, 18) and the artifact id is postfixed with `-lib` (cf. l. 2). The project `packaging` (cf. l. 4) has to be set to `pom` and the dependencies of all artifacts provided by the created library have to be added. The first dependency provides the main binary jar (cf. ll. 6-10), the second dependency (cf. ll. 11-16) provides the symbols artifact (see classifier l. 14), and the third dependency (cf. ll. 17-22) the model documentation with classifier `model-docs`. Since Maven transitively resolves dependencies, this configuration also exports further dependencies defined in the concrete library. Please note that this technique can be also used to combine more than one library. For example, all predefined MontiArc libraries are aggregated within a single aggregator project.

6.8.4 Using a Library

To use a library, simply add its dependency aggregator to the dependencies section of the project's configuration (`pom.xml`). As depicted in Listing 6.44, it has to be added as a dependency with type `pom` (cf. l. 6). This automatically adds the artifacts provided by the aggregator

pom.xml

```
<groupId>$(GROUP_ID)</groupId>
<artifactId>$(ARTIFACT_ID)-lib</artifactId>
<version>0.0.1</version>
<packaging>pom</packaging>
<dependencies>
  <dependency>
    <groupId>$(GROUP_ID)</groupId>
    <artifactId>$(ARTIFACT_ID)</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>$(GROUP_ID)</groupId>
    <artifactId>$(ARTIFACT_ID)</artifactId>
    <classifier>symbols</classifier>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>$(GROUP_ID)</groupId>
    <artifactId>$(ARTIFACT_ID)</artifactId>
    <classifier>model-docs</classifier>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

Listing 6.43: A dependency aggregator for a MontiArc library.

to the class- or modelpath. Afterwards the contained components can be used as subcomponent types and are, e.g., proposed by the autocompletion of the MontiArc IDE.

pom.xml

```
<dependencies>
  <dependency>
    <groupId>$(GROUP_ID)</groupId>
    <artifactId>$(ARTIFACT_ID)-lib</artifactId>
    <version>0.0.1</version>
    <type>pom</type>
  </dependency>
  ...
</dependencies>
```

Listing 6.44: Project dependency configuration to reuse a MontiArc library.

6.9 Distributed Simulation*

Sometimes, a simulation can be executed faster by distributing several components over distinct nodes, i.e., distinct threads, processes, and computers. This is especially the case, if the simulated system contains computationally intensive components which are configured in a parallel

way. The default simulation of a system modeled with MontiArc is designed to be executed in a single thread. Thus, a parallel execution of components is only simulated. To distribute a simulation and achieve real parallel execution, several steps have to be performed:

1. Create a host component model for each distributed node.
2. Generate simulation code for these top-level models.
3. Programmatically configure the generated components using TCP/IP simulation ports.

These steps are explained in the following by means of the already known ABP example. The MontiArc models as well as the Java code created for this example can be downloaded from the MontiArc website[15].

Model and Generate Host Components

Initially, the concrete distribution of the nodes has to be determined and designed. This is performed in the following steps:

1. Create an empty host component model for each simulation node.
2. Replicate each outer port of the original system in one of the host components.
3. Then iterate over the subcomponents of the original system and map each to a node by replicating it in a host component. To do so, use the following rules. First, if more than one subcomponent is mapped to the same host component, they should be directly connected within the original system. Second, if a subcomponent in the original system is connected to an outer port of the system, it has to be replicated in the host component that replicates the connected port (step 2).
4. Remove host components which just contain a single subcomponent with a single incoming port. These subcomponents can be directly instantiated on the simulation node later and must not be wrapped into a host component.
5. Then, iterate over all connectors of the original system. Connectors which connect subcomponents that are deployed to the same node are replicated in the corresponding host component. Other connectors are simply dropped.
6. Create ports with the same type and direction in the host components for each unconnected port of a subcomponent. Connect the outgoing ports of the subcomponents with the outgoing ports of the hosts.
7. Introduce a delay component for each unconnected incoming port of a subcomponent. Then, connect it with the port and the corresponding outer port of the host component. This is needed to prevent timing deadlocks in the simulation (see Section 3.5.5).

Figure 6.45 depicts the resulting host components `DelayedSender` and `ABPReceiver`. The former simply mirrors the interface of the contained `ABPSender` subcomponent that has been developed in this tutorial and adds a delay in front of its `ack` port. The `ABPReceiver` component can be directly reused since it only has one incoming port. Thus, the host component is not needed for the receiver part of the ABP. Please note that we also excluded the `LossyDelayedChannel` components from the distributed system. Since the created top-level models are regular MontiArc components, we now use the MontiArc generator to derive Java simulation classes.

[15]Tutorial: Alternating Bit Protocol, `http://www.monticore.de/languages/montiarc/examples/`.

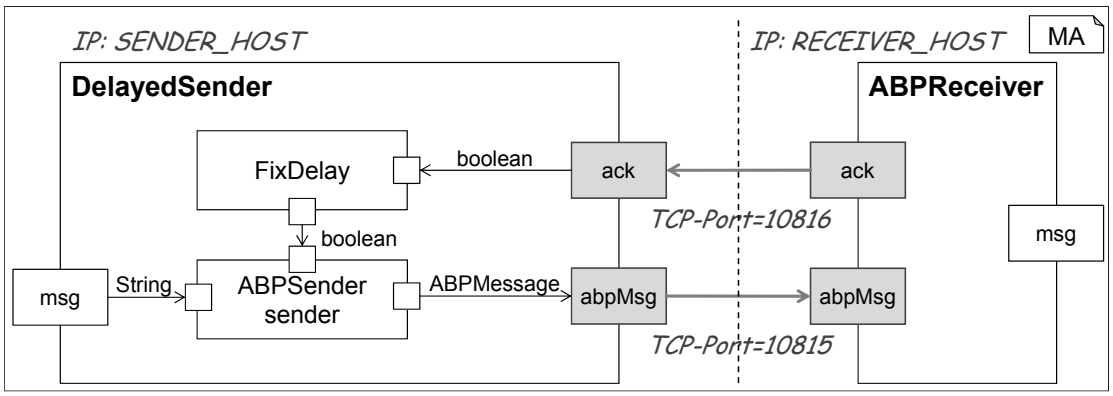


Figure 6.45: A physically distributed ABP simulation.

Configure Top-Level Components

After code generation has finished, we configure the generated Java components to be used in a distributed way. Therefore, an execution class has to be created for every distributed simulation node. This class has to a) configure the node for the distribution and b) execute the configured component. The concrete distribution is also depicted in Figure 6.45. It can be seen that component `DelayedSender` is deployed on a node with IP address `SENDER_HOST` and component `ABPReceiver` is deployed on a node with the IP address `RECEIVER_HOST`. The `abpMsg` ports are configured to communicate via TCP/IP port `10815`, the `ack` ports via TCP/IP port `10816`.

The Java method depicted in Listing 6.46 configures the `ABPReceiver` according to the desired configuration. The following steps are performed:

1. An `ABPReceiver` instance is created using the corresponding factory (l. 6).
2. The simulation scheduler is configured to use a `TCPPortFactory` to instantiate simulation ports and the setup of the receiver is performed (ll. 8 – 12). Consequently, all instantiated ports within the simulation are `TCPPort` objects.
3. A `TCPPort` (an extended simulation port which is capable of communicating via TCP/IP port sockets) is created. By calling its `addReceiver` method, it is configured to send messages to the passed IP address (parameter `addr`) and the given TCP/IP port number `PORT_ACK`. The created port is used as outgoing port `ack` (ll. 14 - 17). Please note that the TCP/IP port numbers are outsourced as constants to a dedicated utility class which is not shown here. See Section 9.2 for more information about `TCPPorts`.
4. The incoming port `abpMessage` is acquired from the component and configured to listen on TCP/IP port number `PORT_ABP` (ll. 19 – 23).
5. To get notified if messages are emitted by port `message`, we also set a `TimedObservablePort` as outgoing port `message` (ll. 25 – 29). The added observer simply prints received messages to the console (ll. 30 - 36).

The sender part of the distributed ABP simulation is configured vice versa. Thus, its outgoing port `abpOut` has to send messages to address `RECEIVER_HOST` on TCP/IP port number

Java

```
/**
 * Configures an ABPReceiver for a distributed simulation.
 * @param addr IP address of the ABP sender.
 */
public void setUp(String addr) {
  receiver = ABPReceiverFactory.create();

  // configure scheduler to create TCPPorts
  IScheduler sched = SchedulerFactory.createDefaultScheduler();
  IPortFactory factory = new TCPPortFactory();
  sched.setPortFactory(factory);
  receiver.setup(sched, new SimpleErrorHandler());

  // configure ack out port
  TCPPort<Boolean> ackOut = new TCPPort<Boolean>();
  ackOut.addReceiver(addr, PORT_ACK);
  receiver.setAck(ackOut);

  // configure and get abp in port
  IInTCPPort<ABPMessage<String>> abpMsg =
      (IInTCPPort<ABPMessage<String>>) receiver
      .getAbpMessage();
  abpMsg.startListenOn(PORT_ABP);

  // configure message out port
  TimedObservablePort<String> results =
      new TimedObservablePort<String>();
  // Use 'result' as outgoing port message.
  receiver.setMessage(results);
  // observer is updated, if a message is received
  results.addObserver(new Observer() {
    @Override
    public void update(Observable o, Object arg) {
      System.out.println("Received: " + arg);
    }
  });
}
```

Listing 6.46: Configuration of the ABPReceiver for a distributed simulation.

`PORT_ABP` and its incoming port `ack` has to listen on TCP/IP port number `PORT_ACK`. Further, it is implemented as a `Runnable` that reads input from the command line in an endless loop. The read strings are then forwarded to the incoming port `accept` of the configured `DelayedSender` instance. Additionally, a $\surd$ is periodically forwarded to the same port to couple simulation and real time. The complete implementation of the sender configuration is given in Listing E.13 on page 335.

To start the distributed simulation, both configuration classes have to be executed on the cor-

responding nodes. The distributed receiver is started by simply instantiating the configuration class and then calling the depicted `setUp` method. The sender is started by instantiating the configuration class, calling its `setUp` method and finally starting the runnable (either directly or by passing it to a dedicated thread).

In this example, the input of the simulated system is simply read from the command line, its output is printed to the console. Input and output has been simplified to focus on the distribution techniques. Nevertheless, the distributed simulation can be used in regular component tests (see Section 6.4) as well as optimization tests (see Section 6.6).

Chapter 7

MontiArc Extension Method

Reuse is a key technique to increase efficiency of the development process. For example, general purpose languages (GPLs) facilitate to reuse well tested code which provides functionality with a well defined interface. Thus, new functions need not to be developed from scratch but can be build on top of existing libraries. The language workbench MontiCore allows to define reusable languages which can be extended and combined to new languages [KRV08]. Beside the extension of the concrete syntax, that is presented in [Kra10], MontiCore also provides mechanisms for semantic language combination [Völ11, HLMSN+15].

By using these features combined with well defined extension points, the MontiArc language and the corresponding tools can be reused and extended. This allows for adjusting the language to a certain domain or to add further language processing tasks, e.g., analysis or metric calculations, while most parts of the language framework can be directly reused in the adjusted language. This is especially needed, if MontiArc's architectural style is either too generic or needs to be adjusted for the current use case (see Section 1.4).

To facilitate the process of language extension and to allow the reuse of as much tooling as possible, this chapter presents an integrated method to extend MontiArc. The method comprises structured steps to extend the language, model processing, MontiArc's runtime environment (RTE), and the corresponding generators. It does not cover the development of a completely different target RTE, since this activity leads to the development of a new dedicated generator. A method for this task is presented in, e.g., [Sch12, Section 5.3].

The method is structured according to the main reasons for extending MontiArc and it supports the following forms of language extension:

- **Model Processing Extension** allows to add new features to the MontiArc language itself. This comprises the integration of new model analyses, metric calculations, or model transformations. This method is described in Section 7.1.
- **Simulation Extension** allows to handle extended syntax in the simulation, to integrate new simulation features, and to extend or adjust simulation scheduling. The corresponding method is discussed in Section 7.2.
- **Language Extension** is used to adjust MontiArc with new model elements or even extend it to a new architectural style. This requires an extension of the syntax and symbol table of MontiArc. These methods are discussed in Section 7.3.

Please note that all examples used in this chapter are available on the MontiArc examples website[1].

[1] MontiArc Language Extension Example Project, `http://www.monticore.de/languages/montiarc/`

7.1 Model Processing Extension

To realize new features for the MontiArc language, e.g., component analyses or metric calculations, the model processing, which is described in Section 5.1, has to be extended. A systematic method to extend model processing of MontiArc models is depicted in Figure 7.1. It comprises the following steps:

1. **Extend Language**: As an initial step the supporting activity `Extend Language`, which is described in detail in Section 7.3, is to be performed. If needed, it enriches the language with elements required for the current extension.
2. **Objective**: Depending on the concrete extension objective, the next activity has to be chosen.
   a) If new model analysis should be added, activity `Add Execution Unit` is to be performed. It is described in Section 7.1.1.
   b) If preprocessings shall be added, perform activity `Add Transformation` which is presented in Section 7.1.2.
3. **Adjust Tooling**: Both activities are followed by a final activity that configures existing MontiArc tools to use the created extension. Since the needed adaptations depend on the current objective, details are given in the corresponding subsections.

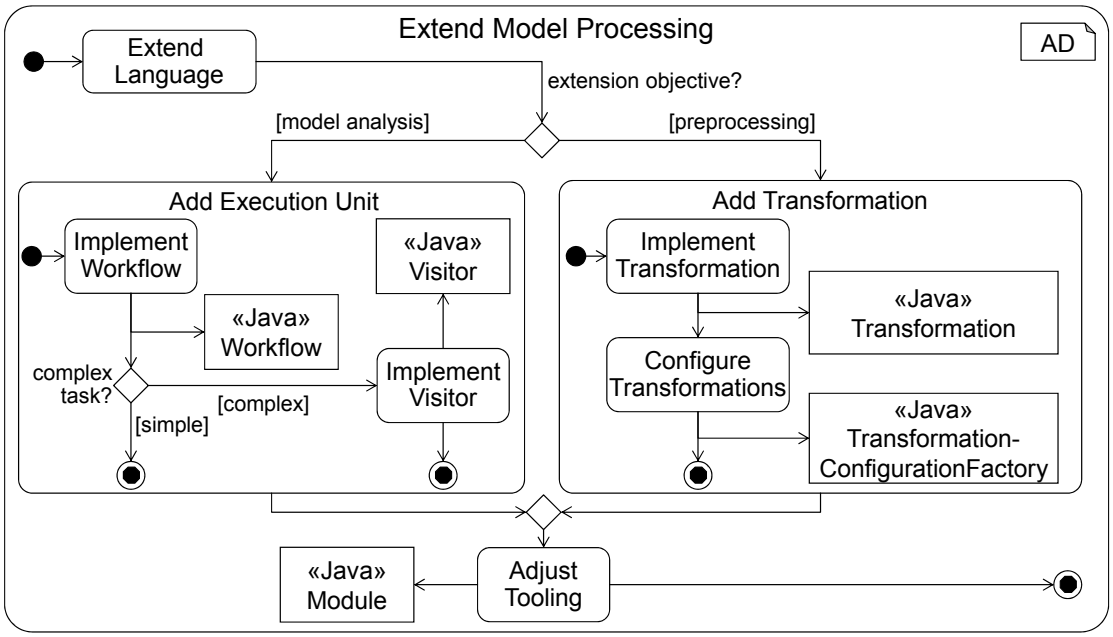


Figure 7.1: Activities to extend the processing of MontiArc models.

examples/.

7.1.1 Add Execution Unit

As already discussed in Section 5.1, model processing tasks are realized using execution units. Thus, new execution units have to be added by registering custom workflows which implement new features at the `ModelingLanguage` component of the language that is to be extended (see Figure 5.1 on page 126). Hence, to extend MontiArc with a new workflow, it has to be added as an execution unit to the registered `MontiArcLanguage` instance (see Figure 5.3 on page 127). Extending MontiArc's model processing is realized by executing the following activities:

Implement Workflow: At first, a new workflow is to be implemented which realizes the new language feature. Therefore, follow [Kra10, Section 9.2] and create a new class that extends class `DSLWorkflow`. The abstract syntax tree (AST) of the processed model is provided by the `DSLRoot` that is passed as parameter to the `run` method of the created workflow. Simple features that depend on a few model elements can be directly implemented in the created workflow. An example that demonstrates how to implement a custom workflow is given in Listing F.2 on page 338.

Implement Visitor: If a complex feature is to be realized, it might by suitable to implement it as a visitor [GHJV95] which simplifies traversing the AST. The MontiCore framework already provides basic classes to realize visitors. As described in [Kra10, Section 9.3.10], new visitors should be realized as `ConcreteVisitor`s. The following three methods can be implemented to visit AST nodes of a certain class. A method parameter with the type of the node class that is to be visited has to be defined.

visit(...): is called before child nodes are visited.
endVisit(...): is called after child nodes have been visited.
ownVisit(...): is called before child nodes are visited. Since automatic traversal of child nodes is suppressed by this method, own traversal strategies of the child nodes can be realized.

To start a concrete visitor, pass an instance together with the start AST node to the static `run` method of class `InheritanceVisitor`. This should be performed in the aforementioned workflow implementation.

Adjust Tooling: To add the custom workflow to the MontiArc tool, a custom guice module[2] has to be created. MontiArc already provides default modules which allow for fine grained modifications while reusing the majority of the default configuration. The default modules depicted in Figure 7.2 are subclasses of `AbstractModule` provided by guice. The contained `bind*` methods are automatically called. Hence, to change a certain injection binding, the corresponding bind method has to be overridden in a subclass. The `additional*Bindings` methods are provided to add further bindings, e.g., needed by new processing workflows. For example MontiArc changes the binding to the component factories which are used for symbol table creation of the ArcD language by overriding method `bindComponentFactories()`. To customize the used `ModelingLanguage`, the method `bindMontiArcModelingLanguage()` has to be overridden. A new `MontiArcLanguage` class is not to be implemented since the workflow can be easily registered at an instance of the existing modeling language.

An example for a custom module is given in Listing 7.3. The aforementioned method is over-

[2] MontiArc uses the guice dependency injection framework (`https://github.com/google/guice`).

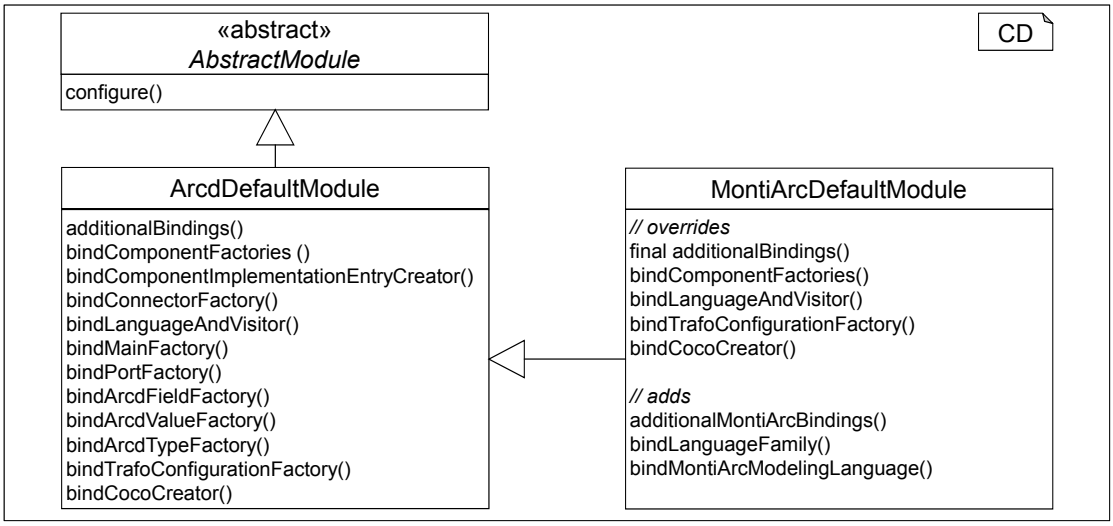


Figure 7.2: Default modules to configure dependency injection.

ridden and left empty to override the default binding (cf. ll. 5 – 8). `ModelingLanguage` instances are provided by the factory method given in ll. 9 – 15 as denoted by the `@Provides` annotation. It creates a modeling language instance (l. 12) using the injected `ILanguage` and registers the custom workflow `PortCountWorkflow` with the execution name `countPorts` (l. 13) that is defined by the constant `COUNT_PORTS` (l. 2). A MontiArc tool instantiated with this module is able to process the created custom workflow.

To execute the registered workflow with the MontiArc tool, it has to be activated in the analysis or synthesis model processing phase (see Figure 5.4 on page 128). For this, suitable ar-

```
1 /** Name of the 'count ports' execution unit.*/                    Java
2 public static final String COUNT_PORTS = "countPorts";
3
4 public static class ProcExtModule extends MontiArcDefaultModule {
5   @Override
6   protected void bindMontiArcModelingLanguage() {
7     // left empty...we simply override default binding.
8   }
9   @Inject @Provides
10  ModelingLanguage getExtendedMontiArcLanguage(ILanguage
11      component) {
12    MontiArcLanguage l = new MontiArcLanguage(component);
13    l.addExecutionUnit(COUNT_PORTS, PortCountWorkflow.create());
14    return l;
15  }
16 }
```

Listing 7.3: Guice module that adds a new workflow to the MontiArc tool.

guments have to be passed to the tool constructor together with an instance of the customized module. An example is given in Listing 7.4. The custom workflow `COUNT_PORTS` is activated in the analysis phase for MontiArc models with the arguments given in l. 7. The MontiArc tool is constructed in l. 10 by passing the defined arguments `toolArgs` with a custom module instance to the tool constructor. The tool is then executed by calling its `run()` method (l. 12).

```
1 // configuration to parse and count ports.                    Java
2 String[] toolArgs = new String[] {
3     "src/main/resources/processing",
4     ARG_OUT, OUT_DIR,
5     ARG_SYMTABDIR, SYMTAB_DIR,
6     ARG_ANALYSIS, MONTI_ARC_ROOT_NAME, WF_PARSE,
7     ARG_SYNTHESIS, MONTI_ARC_ROOT_NAME, COUNT_PORTS
8 };
9 // create tool with custom dependency injection module
10 MontiArcTool t = new MontiArcTool(toolArgs, new ProcExtModule());
11 // execute tool
12 t.run();
```

Listing 7.4: Execute a MontiArc tool with an added workflow.

7.1.2 Add Transformation

To add custom model transformations to the transformation framework which is described in Section 5.3.3, two process steps have to be performed.

Implement Transformation: First, the transformation to add has to be implemented. Therefore, the suitable transformation interfaces which are needed to realize the desired transformation, have to be implemented by a new transformation class. All available transformation interfaces are listed in Table F.4 on page 341.

Configure Transformations: The `TransformationVisitors` registered at the MontiArc tool (see Figure 5.24 on page 148) are set up by a configuration object which is produced by an `ITrafoConfigurationFactory`. Thus, a custom factory has to be implemented that provides a suitable transformation configuration. The relevant interfaces are depicted in Figure 7.5. Transformations might need a symbol table and are executed by a specific execution unit. Hence, the `create` method of the factory gets both as arguments. The name of the execution unit `unitName` can be used to produce different transformation configurations for different phases of model processing. An `ITrafoConfiguration` simply returns a list of transformations which are to be executed by the visitor. They are used by transformation workflows to configure the corresponding transformation visitors. An example for a custom transformation configuration factory, that also reuses the default transformation configuration, is given in Listing F.3 on page 339.

Adjust Tooling: To use the custom factory, it has to be injected by guice. Therefore, a dedicated guice module has to be created that subclasses the default MontiArc guice module and

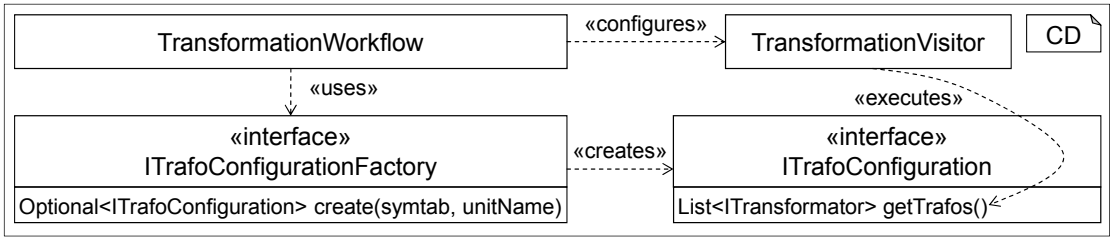


Figure 7.5: Interfaces used as extension points to add further transformations to MontiArc.

overrides its `bindTrafoConfigurationFactory()` method (see Figure 7.2). The overridden method then has to define the binding of interface `ITrafoConfigurationFactory` to the custom factory implementation. The new transformations are executed if the custom module is passed to the MontiArc tool as explained in the previous subsection.

7.2 Simulation Extension

Several reasons lead to the need for an extension of the MontiArc simulation which is described in Chapter 4. For example, new model elements have been introduced which are to be considered in the simulation. A systematic method for this purpose is depicted in Figure 7.6. It comprises the following steps:

1. **Extend Language**: As an initial step, the supporting activity `Extend Language`, which is described in detail in Section 7.3, is to be performed.
2. **Analyze Purpose**: Then, the purpose of the extension is to be analysed. Three main reasons are handled by the method. First, the syntax of the language has been extended and new model elements are to be simulated. Second, a new feature should be added to the simulation. And third, a refined scheduler or a new scheduler strategy is to be realized.

7.2.1 Handle Extended Syntax

The syntax of the language has been extended and new model elements have to be simulated, too. Therefore, it has to be clarified whether added model elements can be semantically represented by existing concepts which are already handled by the simulation. In this case, the extension is implemented by adding a transformation which transforms the new element to a set of semantically equivalent elements. For example, a connector, which is annotated with a delay, can be transformed to two connectors and a delay subcomponent that is interconnected between the original source and target. The activities of `Add Transformation` are described in Section 7.1.2. If a new element is not representable by existing concepts, a new feature has to be added to the simulation.

7.2.2 Add Feature

Features are added to the MontiArc simulation by performing the following activities:

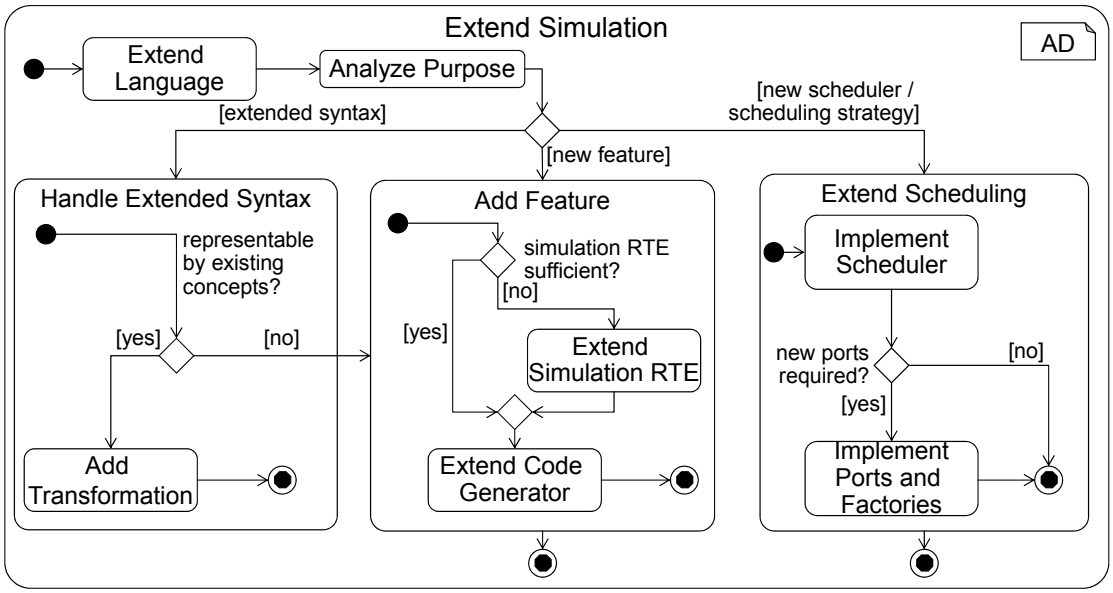


Figure 7.6: Activities to extend the simulation of MontiArc models.

1. **Simulation RTE sufficient?**: Initially, it has to be checked whether the existing simulation RTE is sufficient to integrate the new feature. This is the case if new features can be realized by extending implementation methods which are already created by the MontiArc code generator. Further, the new feature has to only depend on interfaces respectively classes which are already provided by the current simulation RTE.
2. **Extend Simulation RTE**: If the current simulation RTE is not sufficient, it has to be extended. This mostly comprises the following activities:
   - Subtype existing simulation classes to refine generally used abstract implementations, e.g., `AComponent`.
   - Extend existing interfaces to add new methods to existing concepts. For example, further properties like a location or process effort can be added to special components kinds.
   - Introduce new interfaces or classes that represent new elements which are to be simulated. Introducing new RTE classes can lead up to the need for further extensions of the simulation. For example, an automaton interface can be designed that defines methods of the Java representation of an embedded automaton. Since it requires a certain scheduling strategy, component scheduling has to be extended.
3. **Extend Code Generator**: To integrate calls to the added feature into generated component code, the existing code generator has to be extended. This activity is described in detail in Section 7.2.4.

7.2.3 Extend Scheduling

To integrate refined or completely new scheduling strategies into the MontiArc simulation, the following steps have to be performed:

Implement Scheduler: First, the new scheduler is to be implemented by creating a class that implements interface `IScheduler`. Method `setupPort` is responsible to initialize a concrete scheduler with the ports of the scheduled components. The concrete scheduling strategy is to be implemented in method `registerPort`.

Implement Ports and Factories: Scheduling strategies are also represented in the communication between port objects and the scheduler. Therefore, new scheduling strategies might lead up to the implementation of specialized ports which are able to interact well with a new scheduler. In this case, these ports have to be implemented together with a port factory. This port factory has to be used by the new scheduler to instantiate port objects (see Figure 4.12 on page 97).

7.2.4 Code Generator Extension

A structured method to extend the MontiArc code generator is depicted in Figure 7.7. The generator uses the common generator infrastructure provided by MontiCore (see [Sch12, Section 5.2]). The method comprises the following steps:

1. **Create Generator Configuration**: Initially, a configuration has to be created for the new generator. This configuration has to bind hook points of the generator with a set of templates which should be included at the location of the hook point. To create a valid configuration, it is recommended to copy and adjust the default generator configuration `ComponentMain.ftl` which is located in the MontiArc backend jar[3]. Unbound hook points serve as extension points which allow to add new templates without adjusting the existing template structure. A complete list of these extension points is given in Table F.5 on page 342. The names of the hook points determine their location in the generated code. For example, templates bound to the `constructorHook` are included at the end of the generated constructor.
2. **Adjust Generated API**: If the simulation RTE has not been extended in a previous activity, this step can be skipped. If it has been extended, the application programming interface (API) of the generated component class hat to be adjusted accordingly by performing the following activities:
   a) **Adjust Generated Type Names**: The type names of the simulation RTE classes are not directly used within the templates of the generator. They are imported by the template calculator `SimAPIClassNameCalculator` which maps the qualified class name of an RTE class to its unqualified name. For example, the qualified type name of the abstract component implementation is accessed by using the variable `${AComponent}` in a template. Thus, to replace the generally used superclass of the generated component class, the call of the aforementioned template calculator

[3] de.montiarc.be-2.5.0.jar

has to be replaced with a call to an adjusted version which maps variable `AComponent` to the qualified name of the type that is to be used.

b) **Implement Method Templates**: If the API of the simulation RTE has been extended by adding new methods to extended component interfaces, these methods usually have to be implemented by the generated component class. Therefore, new templates which generate these methods have to be created.

c) **Configure New Method/Element Hook**: To integrate the created templates into the generator, its hook point configuration has to be adjusted. If new templates are to be called with an `ASTArcComponent` AST node, they have to be mapped to hook point `newMethodsHook`. If they need an `ASTArcElement` AST node, they have to be mapped to hook point `arcElementHook`. Both extension points are called within the body of the generated component.

3. **Adjust Existing Methods**: Two activities have to be accomplished to adjust existing generated methods.

a) **Implement Extension Templates**: First, templates have to be implemented that produce the code which is to be injected into existing methods. Which AST node type is available in the these templates is documented in the default generator configuration respectively in Table F.5 on page 342.

b) **Configure Corresponding Method Hook**: Finally, the generator configuration has to be adjusted by adding the created templates to the corresponding method hooks.

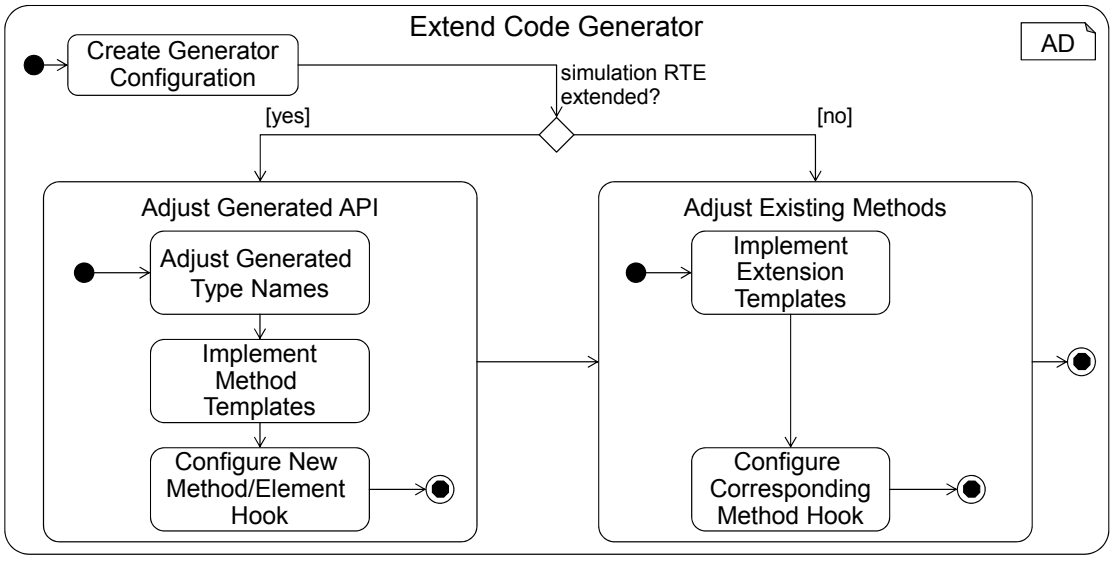


Figure 7.7: Activities to extend the MontiArc code generator.

As described in Table 5.54 on page 175, the adjusted generator can be started by passing the `-generator` parameter triple to the MontiArc generator tool. Please note, all template have to be in the Java class path.

7.3 Language Extension

An extension of the MontiArc language is often a preparatory step for extension of the model processing or the MontiArc simulation. As depicted in Figure 7.8, it comprises the following structured activities:

1. **Analyze Extension**: As an initial task, the main extension is to be analyzed to answer the question whether it depends on existing model elements. If the needed elements already exist in the MontiArc language, an extension of the language is not needed. Thus, this preparatory process is finished.
2. **Extend Syntax**: If needed model elements are not available, the syntax of the language has to be extended as described in Section 7.3.1. Afterwards, it has to be considered whether the extended syntax also affects the symbol table. If this is not the case, the language extension activity is finished.
3. **Extend Symbol Table**: In the case of an influence on the symbol table, it has to be extended as described in Section 7.3.2. An influence is given if added elements should be part of the outer interface of a component (public symbol table) or should be inherited from a supercomponent (protected symbol table).

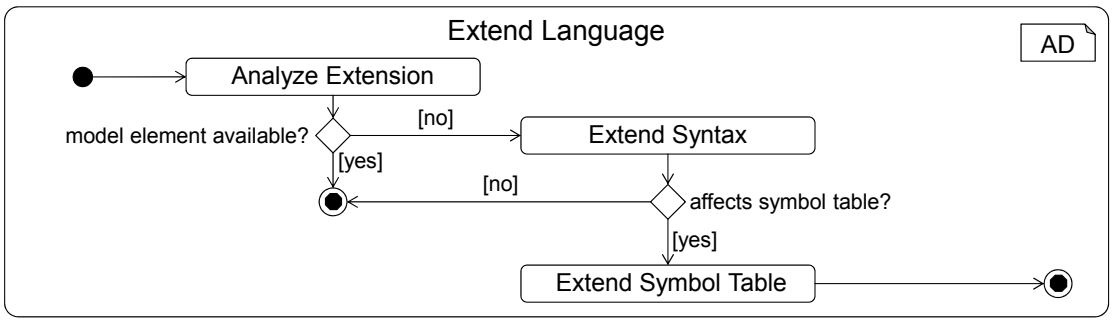


Figure 7.8: Activities to extend the MontiArc language.

7.3.1 Syntax Extension

The definition of new architectural styles based on MontiArc requires an extension of the syntax. Sometimes, also new language features for MontiArc's basic style need more information than the given MontiArc language provides. If, e.g., a metric is to be realized which calculates the mean execution time of decomposed components, it needs the consumed CPU cycles of contained subcomponents. However, in the default MontiArc language, component definitions can not be attached with such information. Thus, its syntax needs to be adjusted.

A structured process to extend MontiArc's syntax is depicted in Figure 7.9. It comprises the following activities:

1. **Analyze Purpose**: MontiArc's syntax can be extended in several ways. Depending on the concrete purpose, a suitable extension method has to be chosen.

a) **Extension by Inheritance** is most suitable when existing language elements should be refined or new language elements are to be added.
b) **Extension by Embedding** is to be used when a new behavior or constraint definition language is to be added to the provided extension points.
c) **Extension by Stereotypes** can be used to annotate existing model elements with further information in a lightweight way.

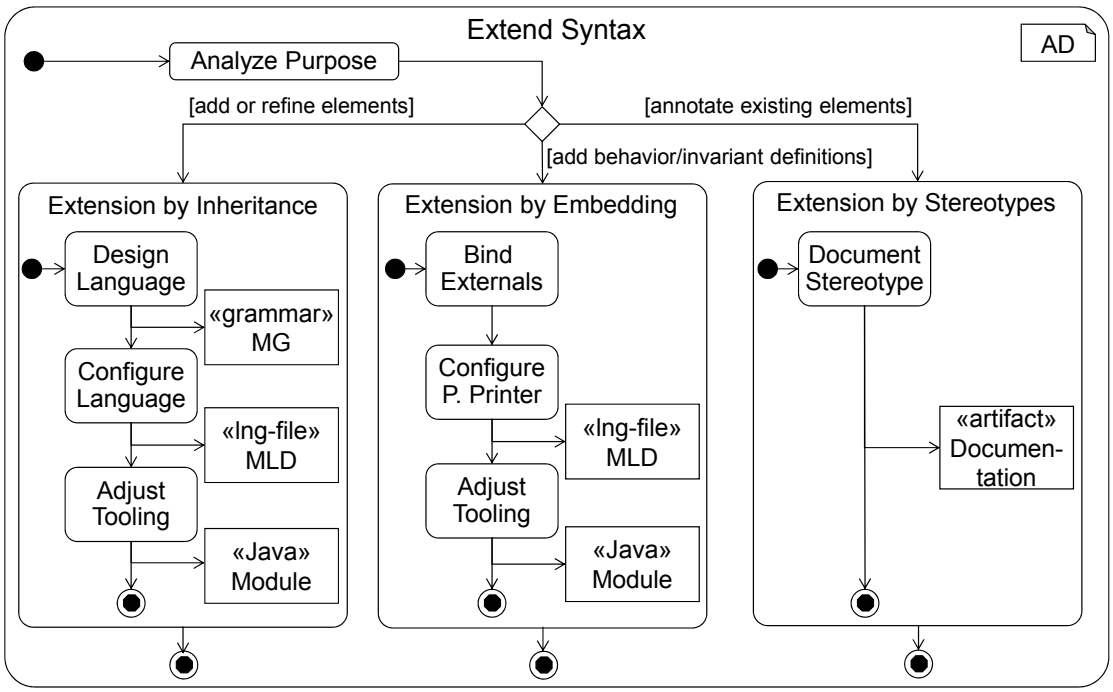


Figure 7.9: Activities to extend the syntax of MontiArc.

Language Inheritance

To adapt the concrete syntax of the language with new or refined language elements, language inheritance is suitable. The Architecture Diagram (ArcD) and MontiArc grammars, which are presented in Section 3.4.3 and Section 3.4.4, provide two interfaces that also serve as extension points for language inheritance. `ArcElement` is to be implemented by each production which defines a nonterminal of a new language element. As a result, such elements can be used everywhere within the body of a component. This also holds for the second interface `MontiArcConfig` which is provided for configuration extensions. While both interfaces do not influence the concrete syntax of the implementing productions, they still affect the types of the resulting AST classes. Consequently, they indicate a certain meaning, i.e., the nonterminal represents an architectural element or a component configuration. Further, configuration and regular element nodes are distinguishable during model processing activities. An example that explains how to create an extension of the MontiArc language is given in the following. Components defined

in the extended language also contain property information for analysis, as, e.g., the component depicted in Listing 7.10.

```
component ExtendedMontiArcComponent {
  // ...
  property delay = 5;
  properties {
    CPU_CYCLES = 10;
    MEMORY_CONSUMPTION = 150;
  }
```

MA[PR] ...

Listing 7.10: Example component which contains extended model elements.

Design Language: First, a grammar has to be created which extends the MontiArc language. All productions that define nonterminals in the subgrammar which should be available within a component's body have to implement one of the aforementioned interfaces. A grammar that adds the desired property definitions is given in Listing 7.11. The nonterminal `SingleProperty` (cf. ll. 2f) represents a single property and the nonterminal `PropertySet` (cf. ll. 5 – 8) a set of properties. Both implement interface `MontiArcConfig` to indicate their semantics of a configuration element. Thus, both can be used within a component's body. Please note that existing language elements can also be replaced or refined in a sublanguage. Therefore, MontiCore's nonterminal inheritance mechanism can be used which corresponds to an alternative in the grammar. Thus, production `A extends B = A_RHS` can be also read as `A = B | A_RHS`. Hence, if `A` should replace `B`, `B` has to be set abstract. A detailed discussion of MontiCore's grammar format is given in [KRV07b] and [Kra10, Chapter 3].

```
1 grammar ExtendedMontiArc extends mc.umlp.arc.MontiArc {
2   SingleProperty implements MontiArcConfig =
3     "property" Property ;
4
5   PropertySet implements MontiArcConfig =
6     "properties" "{"
7       properties:Property*
8     "}";
9
10  Property =
11    Name ("=" Value)? ";";
12 }
```

MG

Listing 7.11: MontiCore grammar which adds properties and property sets as new language elements to the MontiArc language.

Configure Language: Second, a MontiCore language definition has to be implemented that defines the names of tool related classes. An example is given in Listing 7.12. The name of the DSLRoot (cf. l. 3), the DSLRoot factory (cf. ll. 5 – 8), as well as the name of the parsing workflow (cf. ll. 9f) have to be defined (cf. Section 5.1).

```
MLD
1 language ExtendedMontiArcLanguage {
2
3   root ExtendedMontiArcRoot<MCCompilationUnit>;
4
5   rootfactory ExtendedMontiArcRootFactory for
6       ExtendedMontiArcRoot<MCCompilationUnit> {
7     // ...
8   }
9   parsingworkflow ExtendedMontiArcParsingWorkflow for
10      ExtendedMontiArcRoot<MCCompilationUnit>;
11 }
```

Listing 7.12: Basic MontiCore language definition of the extended MontiArc language.

Adjust Tooling: Finally, to reuse existing language processing workflows, such as context condition checks or code generation, the MontiArc modeling language which is used by the MontiArc tool is to be adapted (class `MontiArcLanguage`, see activity **Adjust Tooling** in Section 7.1.1). Therefore, a custom guice module that configures the existing MontiArcLanguage has to be created or extended. An example is given in the appendix in Listing F.1 on page 337. The contained factory method which produces `MontiArcLanguage` instances replaces the root factory by overriding method `getRootFactory()` in a local class (cf. ll. 7-14). To reuse default model processing tasks, such as symbol table creation or context condition checks, the corresponding workflows have to be initialized to use the extended DSLRoot (cf. ll. 18-28). Also the language instance has to be configured to work with the extended DSLRoot instead of the regular MontiArc root (cf. ll. 15-17). By passing the adjusted guice module to the MontiArc tool, it is configured to process extended MontiArc models (cf. Listing 7.4).

Language Embedding

Another possibility to extend MontiArc's syntax is language embedding. According to [KRV08] and [Kra10, Section 4.2], this technique allows to embed parts of another language into a host language at predefined extension points. These are given by external nonterminals. To use such an extension point, one or more bindings to nonterminals of the embedded language have to be defined in the language definition of the host language. MontiArc offers two external nonterminals for this purpose. Nonterminal `BehaviorEmbedding` is used to embed behavior definition languages into the right-hand side (RHS) of `ArcComponentImplementation` (see Listing 3.25 on page 56). `InvariantContent` is intended for the embedding of constraint definition languages into the RHS of `MontiArcInvariant` (see Listing 3.26 on page 56). In the MontiArc language definition, the former extension point is not bound while the latter is bound to Object Constraint Language (OCL) expressions as well as Java block statements. To extend MontiArc with further embeddings, the following steps have to be performed.

Bind Externals: First, a custom language definition has to be created to define the bindings of the targeted embedding as well as the mandatory elements. An exemplary configuration of the root factory which defines the embedding of UML/P statecharts (SCs) (cf. [Sch12, Section 3.3])

into MontiArc is depicted in Listing 7.13. A start parser is defined (cf. l. 3) named `arc` and the `SCDefinition` parser is bound to the external nonterminal `BehaviorEmbedding` (cf. l. 6). Since UML/P SCs itself uses embedded languages to describe actions or pre- and postconditions, the embedded `SCDefinition` parser has to be configured accordingly, too. This is done by the embedding of Java expressions into extension point `InvariantContent` and Java block statements into `Statements` (cf. ll. 9f). Please note that this example further refines the syntax extension given in Section 7.3.1. Nevertheless, the default MontiArc language can be extended by embedding, too. Therefore, a root factory for the MontiArc DSLRoot instead of the extended variant has to be configured in the language definition (cf. l. 2).

MLD

```
1 rootfactory ExtendedMontiArcRootFactory for
2     ExtendedMontiArcRoot<MCCompilationUnit> {
3   MCCompilationUnit arc <<start>>;
4
5   // Embedd statecharts into MontiArc.
6   SCDefinition scp in arc.BehaviorEmbedding(sc);
7
8   // Define statechart embeddings.
9   Expression     javaexpr in scp.InvariantContent;
10  BlockStatement javastmt in scp.Statements;
11
12  // SC, MA, and Java pretty printers.
13  prettyprint {
14    mc.umlp.sc.prettyprint.SCConcretePrettyPrinter;
15    mc.umlp.arc.prettyprint.MontiArcConcretePrettyPrinter;
16    mc.javadsl.prettyprint.JavaDSLConcretePrettyPrinter;
17  }
18 }
```

Listing 7.13: MontiCore language definition of the extended MontiArc language.

Configure Pretty Printer: Then, the pretty printer of the language has to be configured. This enables the resulting pretty printer to also handle the embedded AST nodes. In the given example, this is done by registering the SC, the MontiArc, and the Java pretty printer (ll. 12 – 17).

The depicted language configuration allows to process components which contain SCs to define the component's behavior. Thus, components like the one depicted in Listing 7.14 can be defined. The SC (cf. ll. 8 – 14) is embedded into the `implementation` description of the component (cf. ll. 7 – 15). It defines an initial state `Ping` (cf. l. 8) which is switched to state `Pong` if port `ping` receives a `true` (cf. l. 11). If a `false` is received, it switches from `Pong` to `Ping` and emits a `true` on port `finished` (cf. ll. 12). Please refer to [Sch12, Section 3.3] for a complete definition of the SC language.

Adjust Tooling: Finally, a guice module has to be created which configures the `MontiArcLanguage`. It has to be adjusted to use the root factory that has been defined in the corresponding MontiCore language definition. Therefore, the final activity **Adjust Tooling**, which is described in the previous section, has to be performed. Please note, if further embeddings

```
component MontiArcWithSCComp {                                    MASC
  timing sync;
  port
    in Boolean ping,
    out Boolean finished;

  implementation sc PingPongImpl {
    statechart PingPong {
      initial state Ping;
      state Pong;

      Ping -> Pong: [ping == true];
      Pong -> Ping: [ping == false] / {sendFinished(true);}
    }
  }
}
```

Listing 7.14: Example component which contains an embedded UML/P SC to define the behavior of the component.

are defined for the default MontiArc language, only the root factory has to be replaced with the adjusted one. The model processing workflows need not to be adjusted since they still operate on the default MontiArc DSLRoot. Again, instances of the extended language can be processed by passing the custom module to the MontiArc tool (see Listing 7.4 for an example).

Stereotypes

Additionally, stereotypes can be used to add further semantic information to MontiArc components without an explicit extension of the modeling language. A stereotype can be preceded to component definitions, ports, subcomponents, connectors, as well as behavior definitions. Also stereotypes can be attached to configuration elements like the `autoinstantiate` or the `autoconnect` statement. The defining production is inherited from the MontiCore Commons language collection (cf. Figure 3.17 on page 52, [Sch12, Section 3.8]). A stereotype has the following syntax: `<< (Name (= value:String)? )+ >>`.
However, stereotypes have the drawback that their syntax is not checked by the parser beyond the aforementioned rule. Thus, a mistyped stereotype name will not result in a parser error.

Document Stereotype: According to [Rum11, Section 2.5], stereotypes are used for distinct purposes ranging from controlling the code generator to the documentation of unfinished or underspecified parts of the model. To enrich a new stereotype with an informal semantics, [Rum11] proposes a stereotype definition template which allows to document the motivation, preconditions, the kind of use, and the effect of a stereotype. It is recommended to use and probably extend this template to document a newly introduced stereotype. This documentation can serve as an informal requirement definition for the tool developer to properly handle elements annotated with a stereotype. Additionally, it can be integrated into the documentation which is create for modelers.

7.3.2 Symbol Table Extension

An extended syntax of the MontiArc language can lead up to an extension of the symbol table. A structured method for this purpose is depicted in Figure 7.15. It comprises the following activities:

1. **Analyze Cause**: First, the cause which leads up to the symbol table extension needs to be analyzed. Based on the result, one of the next activities is to be performed.
2. - **Adapt Type Language**: Is suitable to integrate a new type definition language into the MontiArc tooling.
   - **Add Behavior Entry**: Is to be performed to represent behavior definitions within the symbol table which have been defined in an embedded behavior definition language.
   - **Add Entry**: Is suitable to integrate new model elements into the symbol table of MontiArc.
   - **Refine Entry**: Is used to refine existing entries for refined model elements.
3. **Adjust Tooling**: Finally, depending on the concrete symbol table extension, the MontiArc tooling has to be adjusted accordingly.

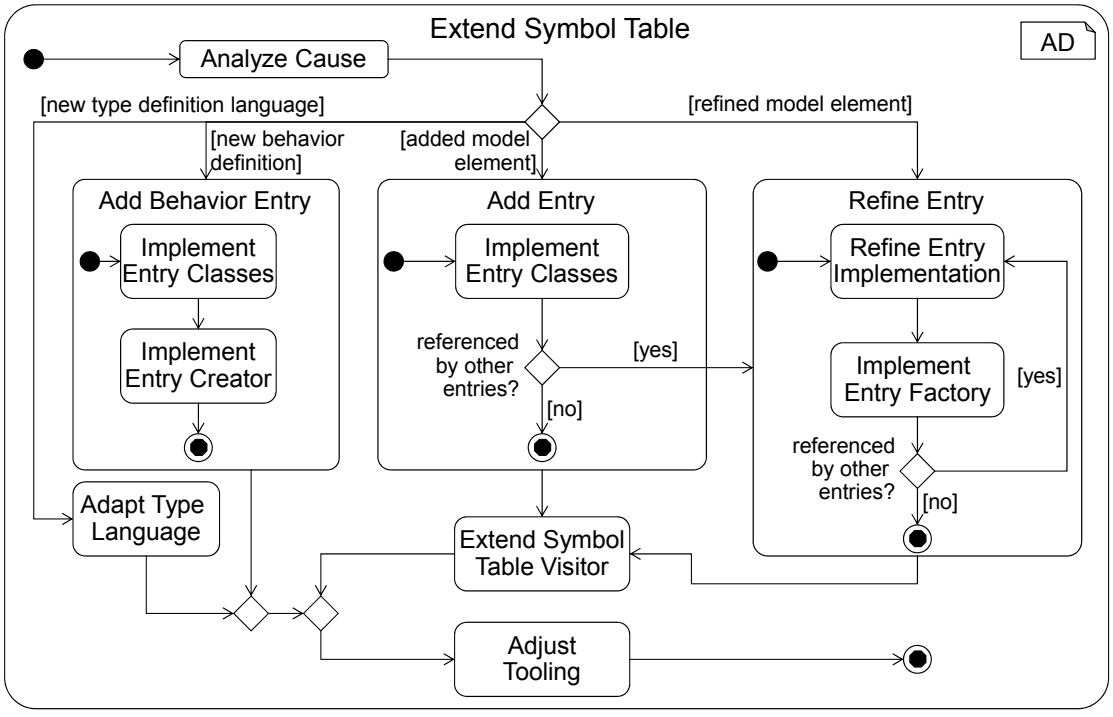


Figure 7.15: Activities to extend the symbol table of MontiArc.

Adapt Type Language

To integrate a new type language into the MontiArc tooling, an aggregation of the MontiArc language and the type language has to be performed. This way, types defined in the language

to integrate can be referenced within MontiArc models. The symbol table techniques to realize language aggregation are described in detail in [Völ11, Section 7.4]. The following steps are needed to integrate a new type language into the symbol table:

1. **Adapt Entries**: The symbol table entries from the type language have to be translated to the entry `ArcdTypeEntry` (see Figure 5.8 on page 135). An adapter [GHJV95] is suitable to perform this translation by converting an interface of a class into another interface which is expected by a client. In case of MontiArc, the client corresponds to the symbol table that handles `ArcdTypeEntries`. Thus, the adapter has to translate the interface of the type entry which is to be integrated into the interface of an `ArcdTypeEntry`. Please note that the term interface in this context can be also interpreted as the set of methods provided by a certain class. Hence, a suitable adapter does not necessarily translates Java interfaces. An example is given in Figure 7.16. Class `CDType2ArcdTypeAdapter` translates calls to the methods provided by class `ArcdTypeEntry` to methods provided by the `adaptee`.
2. **Create Qualifiers**: MontiArc's type entries are created either in a unqualified or qualified state in the symbol table construction process. Unqualified entries are adapted during entry qualification by a dedicated `IQualifierClient`. Already qualified entries are adapted afterwards by an `IQualifiedEntryHandler`. Both discover the corresponding adaptee, create an adapter object, and set the adapter as qualified version of the adapted entry.
3. **Create Resolvers**: In addition, new resolver clients are needed to resolve elements of the adapted type language. These are responsible for resolving elements of the MontiArc language, e.g., an `ArcdTypeEntry`. Internally, they use the registered resolver for the adapee, e.g., a CD type resolver, and return an adapted entry.
4. **Adjust Tooling**: The created qualifier and resolver components need to be registered at the language family which is used by the MontiArc tool. A suitable way is to create a specialized LanguageFamily provider which internally uses the `MontiArcLanguage-FamilyFactory` provider and adds the new components to the created object. To use the new provider, override method `bindLanguageFamily` of the default MontiArc guice module and bind the provider to the `LanguageFamily` class.

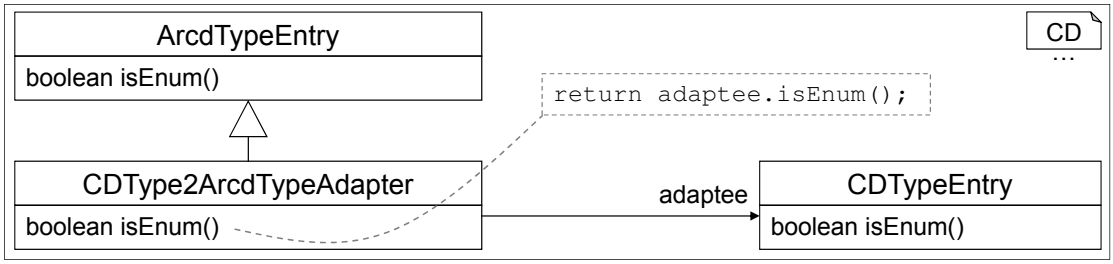


Figure 7.16: Adapter to translate CD type entries to ArcD type entries.

Add Behavior Entry

The MontiArc symbol table serves a dedicated extension point to integrate entries for behavior definitions without creating a new symbol table visitor. To use this extension points, the following steps have to be performed:

1. **Implement Entry Classes**: A MontiArc component entry can contain component implementation entries (see Figure 5.8 on page 135). Depending on the embedded language, entries have to be implemented which represent the embedding in the MontiArc symbol table. If the embedded language provides its own symbol table, the contained entry classes can be reused by adapting them to the representing entries. A concrete component implementation entry then aggregates the adapters or the dedicated entries. Additionally, resolvers, qualifiers, and deserializers have to be realized which handle the created symbol table elements.
2. **Implement Entry Creator**: To integrate these entries as well as the supporting infrastructure into MontiArc's symbol table, an `IComponentImplementationEntryCreator` has to be implemented. It is responsible for constructing the implementation entries, integrating them into the symbol table, and providing the corresponding resolvers, qualifiers, and deserializers. By extending the abstract class `AComponentImplementationEntryCreator`, only entry creation has to be realized.
3. **Adjust Tooling**: The symbol table visitor delegates implementation entry creation to a registered manager. The manager then decides based on the currently visited AST node which implementation entry creator is to be used. As depicted in Listing 7.17, the creators are then integrated by overriding method `bindComponentImplementationEntryCreator` in a custom guice module. Further, a provider method which returns a configured implementation creator manager has to be implemented. Afterwards, the new implementation entries are available in the symbol table of processed components.

```java
@Override
protected void bindComponentImplementationEntryCreator() {
    // overridden to remove default binding
}

@Provides
protected IComponentImplementationEntryCreatorManager
    createImplCreationManager() {
  ComponentImplementationEntryCreatorManager result = new
      ComponentImplementationEntryCreatorManager();
  result.addCreator(new ArcSCImplementationEntryCreator());
  return result;
}
```

Listing 7.17: Integration of implementation entry creators into the MontiArc tool.

Add Entry

Since new language elements have to be present in the symbol table as well, e.g., to allow for context condition checking, representing entries have to be integrated. This is realized by performing the following steps:

1. **Implement Entry Classes**: As a first step, the entry itself has to be implemented. Additionally supporting infrastructure like qualifiers, resolvers, and deserializers for the new entry have to be developed.
2. **Extend Symbol Table Visitor**: Then, MontiArc's symbol table visitor has to be extended in a subclass. The extended visitor has to implement visit methods for the newly added language element. There, a new entry has to be constructed and added to the symbol table in the corresponding scopes. To alleviate further extensions of the language, the entry should not be created directly but by a dedicated factory. The factory instance could be injected into the visitor by passing it as an additional constructor argument. If the new entry shall be referenced by another entry, e.g., by a component entry, the referencing entry has to be refined as described in the next section.
3. **Adjust Tooling**: The extended symbol table visitor as well as the supporting infrastructure have to be integrated into the MontiArc tool. Therefore, method `bindLanguageAndVisitor` has to be overridden in a dedicated guice module. Class `ConcreteASTAndNameSpaceVisitor` has to be bound to the extended visitor class. Similar to Listing 7.17, the binding to class `ILanguage` could be realized by a provider method which creates a new `ArcdLanguageComponent` and adds the created qualifiers, resolvers, and deserializers. If a factory has been implemented which is responsible for entry creation, an additional binding for the factory has to be defined. It can be defined in an overridden `additionalMontiArcBindings` method. Afterwards, the created entries can be resolved in the name spaces of the MontiArc symbol table.

Refine Entry

Refining an existing entry implementation is needed if the corresponding language element has been refined or if it should now contain a reference to a newly added language element. In both cases, the following activities have to be performed:

1. **Refine Entry Implementation**: First, the entry class which corresponds to the refined language element (base entry) has to be subclassed. In the subclass, the new references or information have to be stored. Also qualifiers, resolvers, and deserializers have to be developed for the refined entry. Please note that the deserializer can reuse functionality which is provided by the deserializer of the base entry. To reuse existing model processing workflows which are based on the symbol table, e.g., context condition checks, it has to be regarded that:
   - The refined entry has its own kind and does not reuse the kind of the superclass. This also enables to develop dedicated context conditions for refined elements.
   - An adapter is implemented which translates a refined entry into its base version.
   - A resolver should be provided which resolves a refined entry if it is asked to resolve a base entry.

2. **Implement Entry Factory**: All entries of the MontiArc symbol table are instantiated by dedicated factories. Thus, a factory has to be created which implements the interface of the factory that is responsible for producing the base entry. If, e.g., an extended component entry is developed, a factory has to be created which implements the `IComponentEntryFactory` interface or subclasses the original factory class.
3. Afterwards, it has to be checked whether the base entry has been referenced by another entry. In this case, the referencing entry has to be refined, too. If, e.g., a special component entry has been created, the subcomponent entry has to be refined to provide the new entry as component type instead of a regular component entry.
4. **Extend Symbol Table Visitor**: To integrate the information provided by refined model elements into the symbol table, the creating visitor has to be extended in a subclass. New visit methods have to handle the refined model elements by creating refined entries, setting them up, and adding them to the corresponding symbol tables, referencing entries, as well as scopes.
5. **Adjust Tooling**: Finally, the tooling has to be adjusted to use the refined factories, the extended symbol table visitor, as well as the created support classes. To replace the existing factory, the corresponding method of the default guice module has to be overridden to bind the factory interface to the class of the extended factory. How to integrate the extended visitor and the support classes is described in subactivity `Adjust Tooling` within activity `Add Entry`. Afterwards, MontiArc's symbol table contains all refined entries and is able to translate between base entries and their refined variants.

Chapter 8

Case Studies Using MontiArc

The MontiArc architecture description language (ADL) with the corresponding simulation framework is mainly defined and developed by the Software Engineering workgroup Aachen. In this chapter, case studies are listed which have been conducted to evaluate the capability of the framework to develop, simulate, and test distributed information flow architectures. The results of these case studies have been used to further develop and refine the MontiArc framework to increase its usability. Section 8.1 gives a brief overview of conducted case studies. Two of them are presented in more detail. First, the development of components to model and simulate TCP/IP communication. Second, MontiArc is used to derive FlexRay clusters from logical information flow architectures.

8.1 Overview

The following case studies have been conducted to evaluate the MontiArc ADL and the simulation framework:

- *Monitoring Online Systems Using MontiArc*. In [Ix12], reusable components have been developed to monitor external services as well as properties of the computer on which they are executed (agent components). This case study has been mainly conducted to evaluate MontiArc's usability as an architectural programming language (APL) (see requirement *LRQ2*). Therefore, the simulation runtime environment (RTE) has been extended with remote ports which have been originally developed for AJava (see Section 9.2). These ports are able to communicate using regular TCP/IP connections to connect distributed monitoring agents. A central controller then collects incidents which are reported by the agents, creates, and sends reports, e.g., per e-mail or an instant messenger. The developed components are available in the MontiArc Monitoring Library.

- The *MontiArc Library* has been initially developed in a lab course[1]. It offers a collection of reusable components for image processing, digital techniques, mathematical computations, Internet services, and simulation specific components, e.g., delay or message loss components. A brief description is given in Section 6.8.2.

- *Implementation and Comparison of Distributed System Case Studies Using MontiArc*. A collection of pre-existing case studies, which have been conducted with FOCUS [BDD+93,

[1] Bachelor Lab Course: Simulation Component Development for MontiArc, `http://www.se-rwth.de/teaching/ss10/montiarc/`.

BS01], AutoFOCUS [BHS99, HF10] or Ptolemy [EJL+03], are re-implemented using MontiArc in [Kau13]. The thesis comprises a Modulo8-counter (FOCUS, [Fuc95]), an elevator control system (AutoFOCUS, [SW99]), a traffic lights controller (Ptolemy II, [BFLvH08]), a Railroad Crossing and an assembly line (Ptolemy II, [BOF+10]). The implementation of the case studies is partially available on GitHub[2].

- *Modeling and Simulation of the TCP/IP Stack* conducted in [Sch13]. Details are given in Section 8.2.
- *Model-Driven Development and Simulation of FlexRay-Based Systems Using MontiArc* conducted in [Rab13] is presented in Section 8.3.
- *Interior Light Arbiter*. The logical architecture of an arbiter used to control the interior light of a car[3]. This case study is used as a running example for the MontiArc chapter in the Generative Software Engineering lecture. It has been extended with deltas to model and compare variants of the system using Δ-MontiArc (see Section 9.1) in a project for the Daimler AG.
- The *Alternating Bit Protocol* is realized in a small case study which demonstrates the most aspects of the MontiArc framework, i.e., modeling, behavior implementation, black- and white-box testing, documentation, and component library design. It is used as a running example in Chapter 6.

8.2 Modeling and Simulation of the TCP/IP Stack

MontiArc, as presented in this thesis, allows to simulate the logical communication between clients of a distributed system to predict their interaction in early phases of system development. In the technical deployment of a system, components interact using communication protocols that embed these logic messages into suitable data structures which are then transmitted over the hardware communication link. If this physical connection fails, e.g., the WiFi disconnects or a server does not correspond in time due to heavy traffic, fallback communication strategies have to be implemented into the components of a distributed system. To be able to simulate such scenarios, Transmission Control Protocol (TCP) / Internet Protocol (IP) stack components have been developed using MontiArc in the bachelor thesis *Development and Implementation of the TCP/IP Stack in MontiArc* [Sch13]. These components simulate a selected set of protocols from the TCP/IP stack, one of the most important protocol stacks which builds the basis of the Internet [FS11, Chapter 1]. The created components and implementations are available on GitHub[2].

8.2.1 The TCP/IP Stack - An Introduction

According to the Internet Engineering Task Force [For89], the TCP/IP stack consists of a set of layered network protocols which enable data communication in a local area network (LAN) or

[2]MontiArc examples on GitHub: `https://github.com/arnehaber/montiarc-examples`
[3]MontiArc Example Projects, `http://www.monticore.de/languages/montiarc/examples/`.

between multiple connected LANs. As depicted in Figure 8.1, the stack is hierarchically structured in four different layers which build upon and abstract from a concrete physical network. The four layers are:

1. The **Application Layer** provides means to communicate between two nodes independent from their location, address, and network. The layer is responsible for interchanging application data which is produced or consumed by application programs.
2. The **Transport Layer** realizes addressing within a single node. A unique port is assigned to all programs which communicate via TCP/IP. When a node receives data, these ports allow to map which data has to be relayed to which program.
3. The **Internet Layer** is responsible for passing data between different LANs. Therefore, every participating node is allocated with an IP address that enables targeted sending of data. Based on this layer, routers are able to forward data to a specific receiver without overloading the network with flooded data.
4. The **Data Link Layer** provides functions to send and receive data over logical links. It serves as an abstraction to the concretely used hardware connection to allow an hardware independent realization of the upper TCP/IP layers. Its main task is to encode data into rising and falling potentials which are transmitted via the physical line. Additionally, it ensures that physical signals are not corrupted by avoiding collision with other signals.

8.2.2 TCP/IP Stack Layers in MontiArc

The four layers of the TCP/IP stack contain a set of protocols which allow to transmit different kinds of data. In this bachelor thesis, a selected subset of these protocols has been realized in MontiArc to simulate HyperText Transfer Protocol (HTTP) connections. Thus, the layers are

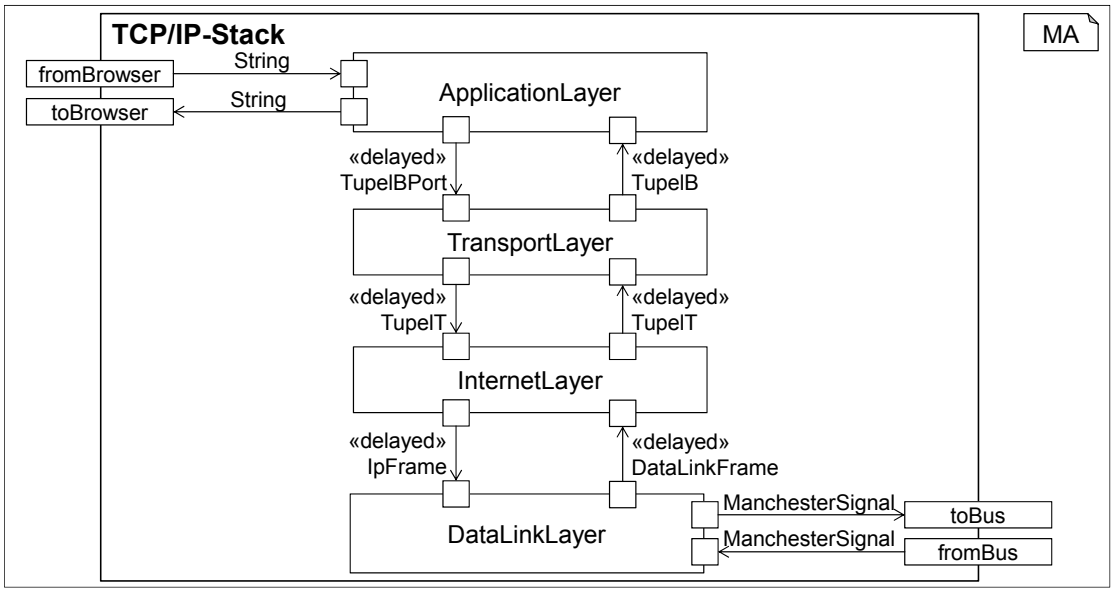


Figure 8.1: Overview of the TCP/IP stack realized in MontiArc (adapted from [Sch13]).

realized as decomposed components and the contained subcomponents instantiate protocol components. An overview of the realized layer components is given in Figure 8.1. The layers are connected bidirectional to build up the TCP/IP stack protocol. The layers are pairwise separated using `FixDelay` components to avoid simulation deadlocks (see Section 3.5.5). It can be connected to a browser application to receive HTTP requests. The resulting `ManchesterSignals` encode the transmission over a physical connection (ports `toBus`, `fromBus`), where, according to the IEEE 802.3 standard [IEE12], a rising edge encodes a logical `one` and a falling edge a logical `null` (see, e.g., [For00]). The resulting HTTP response is transmitted back to the connected application.

The **Application Layer** contains a single subcomponent which instantiates the HTTP component. The architecture of the latter is depicted in Figure 8.2. The target IP address of HTTP requests is resolved by the `Dns` subcomponent which implements the Domain Name System (DNS). Please note that for convenience, the IP address is resolved using a simple lookup table stored in a file which is loaded from the classpath. `GenerateHttpRequest` generates a HTTP-GET-Request. Its payload is encoded to an UTF-8 conform byte array paired with the target port number into a `TupelBPort` object by component `Utf8Encode`. The result is transmitted to the transport layer via port `toTransport`. Replies are received via port `fromTransport` and the payload is initially decoded by component `Utf8Decode`. Component `ReplyBroker` then decides whether the reply is a HTTP-GET-Request or a HTTP-RESPONSE. The former is handled by component `GenerateHttpResponse` and is afterwards encoded and transmitted (see above). Responses are interpreted by component `Interpreter` which drops the HTTP-RESPONSE header and emits the contained HyperText Markup Language (HTML) document via port `toBrowser`.

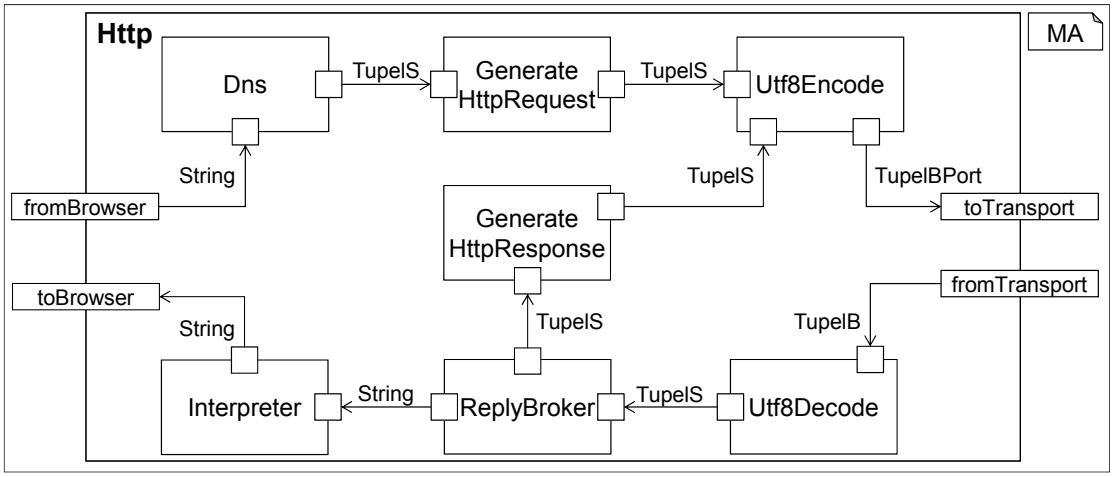


Figure 8.2: Architecture of the HTTP protocol in MontiArc (adapted from [Sch13]).

The **Transport Layer** contains a single subcomponent which implements the Transmission Control Protocol (TCP). The architecture of the `Tcp` component is omitted for reasons of space. However, it performs the following steps and activities. `TupelBPort` objects received from the Application Layer are encapsulated into TCP transmission frames which are then fragmented

into 1500 Byte parts. A control component is responsible to set up or tear down stateful connections to the target and to transmit the fragmented TCP frame to the underlying Internet Layer after a checksum has been added. Received data from the underlying layer is validated by comparing the contained checksum with the calculated checksum of the payload. Depending on the contained port number, the payload is forwarded to the target application after the TCP frame has been removed.

The **Internet Layer** contains a single subcomponent which implements the Internet Protocol (IP). Again, the architecture of the `Ip` component is omitted. The received data from the Transport Layer is first encapsulated with an IP frame. Since data fragmentation within the IP protocol has not been realized in this thesis, data is always marked as *not fragmented*. Finally, it is transmitted to the underlying Data Link Layer. Received data from the underlying layer is first validated by comparing the transmitted checksum with a calculated checksum. Afterwards, the data is passed to a stub component which is responsible for assembling fragmented data. Since fragmentation is not supported yet, it simply forwards the data to another component that removes the IP frame and relays the payload to the upper Transport Layer.

The **Data Link Layer** depicted in Figure 8.3 is realized using `Arp`, `Rarp`, and `Ethernet` components. Component `Arp` implements the Address Resolution Protocol (ARP) which translates the target IP address to a media access control (MAC) address. The Reverse Address Resolution Protocol (RARP), which is implemented by component `Rarp`, translates MAC addresses back to IP addresses (see [For82]). Similar to component `Dns`, this translations are realized using a lookup table located in a file which is loaded from the classpath. The contained `Ethernet` component is decomposed into three layers itself:

- The **Logical Link Control Layer** encapsulates IP frames into Ethernet frames and relays them to the underlying MAC layer. Data received from the underlying layer is validated and the Ethernet frame is removed before forwarding the data.
- The **Media Access Control Layer** controls the access to the underlying transport channel and negotiates access to the channel with other nodes. Collision detection and transmission is realized using the persistent Carrier Sense Multiple Access/Collision Detection (CSMA/CD) approach as defined in standard IEEE 802.3.

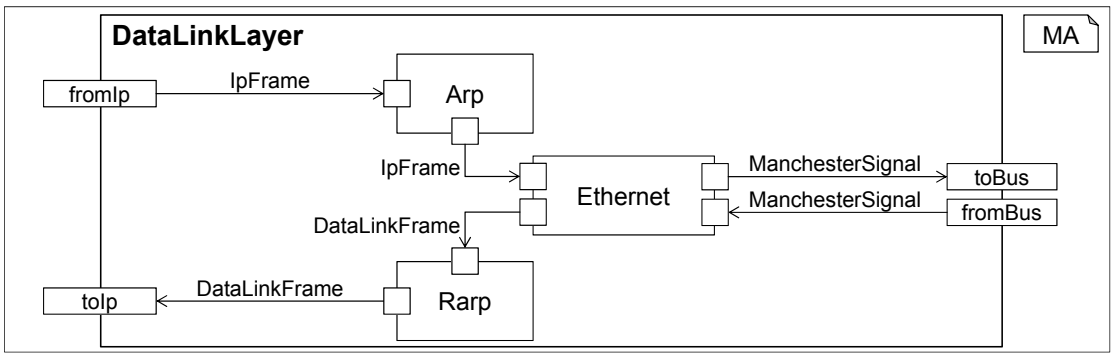


Figure 8.3: The data link layer that is realized using a combination of Arp, Rarp, and the ethernet protocol (adapted from [Sch13]).

- The **Physical Signaling Layer** translates data received from the MAC layer into physical signals by encoding it into Manchester signals. Vice versa, received Manchester signals are decoded to data which is then forwarded to the MAC layer.

8.2.3 Conclusion

In the discussed bachelor thesis a total of 13 decomposed and 27 atomic components have been developed with an average of 2.8 ports, 7 connectors, 3.8 subcomponents, and 13.7 lines of code (without comments) per component. The maximal depth of the component hierarchy is 5. The separated layers and their contained subcomponents have been tested from bottom to top by defining 45 I/O-Tests (see Section 6.4.1) with a total number of 148 tests which cover about 95 % of the manually written code.

Summing up, this case study shows that MontiArc can be used as an APL to develop complex communication protocols. Due to the time limitation of bachelor theses, parts of the TCP/IP protocol stack are abstracted to concentrate on the functionality of other parts of the stack. DNS, ARP, and RARP are realized by simple lookup tables and message routing is omitted. Nevertheless, the presented case study is a base for a MontiArc TCP/IP protocol library which can be stepwise completed with more protocols from the application layer, e.g., Telnet or FTP. If a certain designated functionality is not completely realized, it is encapsulated into a dedicated stub component which already defines the interface of the function. In this way, stub components can be easily exchanged with realistic component implementations to achieve a full-featured TCP/IP stack component library.

8.3 FlexRay Communication Simulation Using MontiArc

The complexity of automotive systems is rising constantly. According to Reichelt et al. [RSG+08], there are about 80 interacting control units in a modern car. Since a pairwise connection between these control units using direct lines would result in an increased weight, usually bus systems are used to connect control units. Nowadays, the most used bus system is the CAN-bus which has been developed in 1983 by Bosch [Rei12]. The greatest disadvantage of the CAN bus is the non-deterministic message transmission [Hei12]. Depending on the priority of other communication participants, bus access may be granted or not. This renders the CAN bus unsuitable for future applications, e.g., X-by-Wire systems such as Steer- or Brake-by-Wire [WNS+05], in which a reliable message transmission with hard deadlines is vital. Since the FlexRay bus is ten times faster (up to 10 Mbits/s), has a guaranteed latency, and is error tolerant, it can be regarded as the successor of the CAN bus. However, according to Heinz [Hei12], a drawback is a more complex configuration and the associated higher development costs. Each node has to be configured with 32 local parameters and 39 global parameters have to be adjusted for each FlexRay cluster.

To allow for a simple, fast, and experimental simulation of logical systems that should be deployed to FlexRay networks, a MontiArc FlexRay component library has been developed in the diploma thesis *Model-Driven Development and Simulation of FlexRay-Based Systems Using MontiArc* [Rab13]. To achieve this goal, the following steps have been performed:

1. Initially, the logical architecture of an (abstracted) adaptive cruise control (ACC) system has been developed. It served as a running example to validate the developed approach. This example is described in Section 8.3.2.
2. FlexRay components have been developed in MontiArc which allow to simulate FlexRay communication for this example. It has been distinguished between core and specific components. The former implement core FlexRay functionality that can be reused in any other system that is to be simulated. The latter are needed to connect the components from the specific example to the core FlexRay components. This activity is presented in Section 8.3.3.
3. Finally, actions have been identified that are needed to transform a logical architecture into an architecture in which logical components are deployed to FleyRay nodes. Then, a generator has been implemented that interprets stereotypes within a logical MontiArc model and derives the needed specific FlexRay components to simulate a concrete deployment for arbitrary MontiArc models. A brief description of the generator and its configuration is given in Section 8.3.4.

8.3.1 FlexRay Introduction

FlexRay is a fieldbus system which supports multiple master nodes within point-to-point, star, and bus connection topologies [Hei12]. It technically supports data rates from 2.5, 5, and 10 Mbit/s per channel. Two channels (channel A and B) can be used to either increase the bandwidth or transmit data in a redundant way.

An overview of a FlexRay communication node is depicted in Figure 8.4. Each transmission channel is accessed using a `Busdriver` which represents the first layer of the FleyRay protocol. Since the second transmission channel is optional, only one `Busdriver` is mandatory. The concrete application is executed on the `Host Controller`. The `FlexRay Communication Controller` implements the FlexRay protocol using a state machine which is

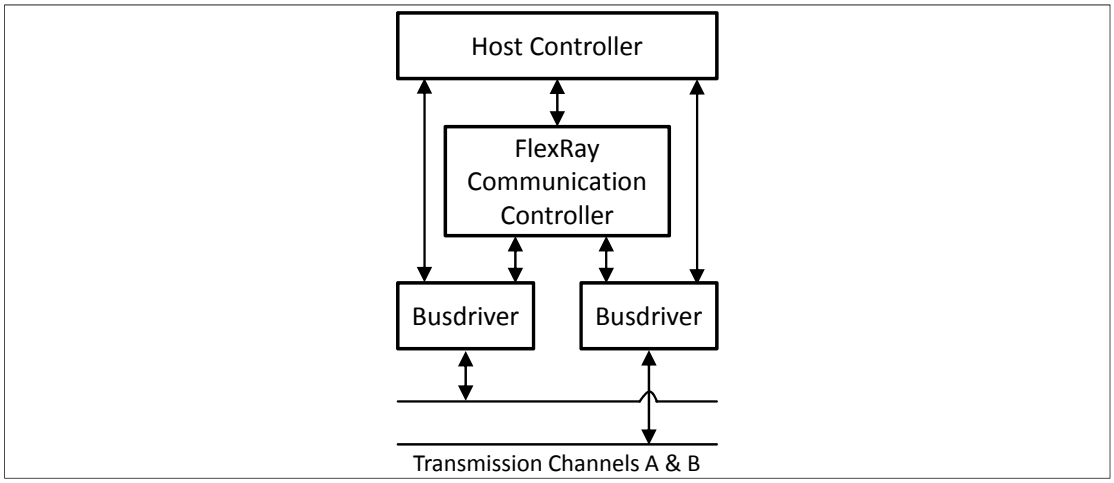


Figure 8.4: Overview of a FlexRay node (adapted from [Hei12]).

called the protocol engine [Hei12]. Additionally, it provides a host specific Controller-Host-Interface that contains the FlexRay parameters of the node and is accessed by the host controller to exchange data with other nodes. Three different node kinds exist. Sync nodes are allowed to send sync frames which are used for clock synchronisation. Coldstart nodes initialize the start of a node cluster by sending startup frames. Normal nodes have no specific properties.

FlexRay systems are organized in clusters that contain several nodes. Each node in a cluster has a unique number. Three different cluster kinds exist which are differentiated by the used synchronisation method and the composition of the contained nodes. The diploma thesis supports the TT-D and TT-L cluster (see [Con10, Section 1.10]). The former consist of two to 15 coldstart and sync nodes with an unrestricted amount of normal nodes. Such a cluster is more robust against faulty nodes. However, a longer startup is needed and the complexity is higher. The TT-L cluster contains a single coldstart node only, which reduces the system complexity. To simulate further coldstart nodes, it emits a second startup frame during the system initialisation.

FlexRay communication is divided into 64 cycles with a configurable length between 16 μs and 16 ms. The cycle length is defined by the cluster. In the simulation, communication streams are separated into time intervals which represent a nanosecond (see Section 4.1). Thus, a communication cycle in the simulation ranges from 16.000 to 16.000.000 time intervals. This time resolution is needed to support the high sampling rates of the FlexRay protocol. As depicted in Figure 8.5, each frame consists of four different segments. The `Static Segment` is divided into up to 1023 slots, while each slot has a fix mapping to a node from the cluster. Assuming that this mapping has been configured correctly, the static segment guarantees that every node of a cluster is at least once able to communicate during a cycle. Messages which are transmitted in the static segment have a fix payload of maximal 254 byte. The payload is simulated using class `CommunicationData` which holds a bit array that represents the payload. In contrast to the static segment, message transmission is not guaranteed in the optional `Dynamic Segment`. It is divided into up to 2047 mini-slots. Further, a single sender can transmit its payload within multiple mini-slots in this segment. Since nodes transmit data in order of their number, nodes with a low number have a higher priority and can block the complete dynamic segment with message transmissions. The optional `Symbol Window` is designed to transmit predefined symbol messages such as collision avoidance, media test, or wakeup symbols. No data is transmitted within the following `Network Idle Time`. It is used by the communication controllers to synchronize their clocks by adjusting the offset and frequency parameters.

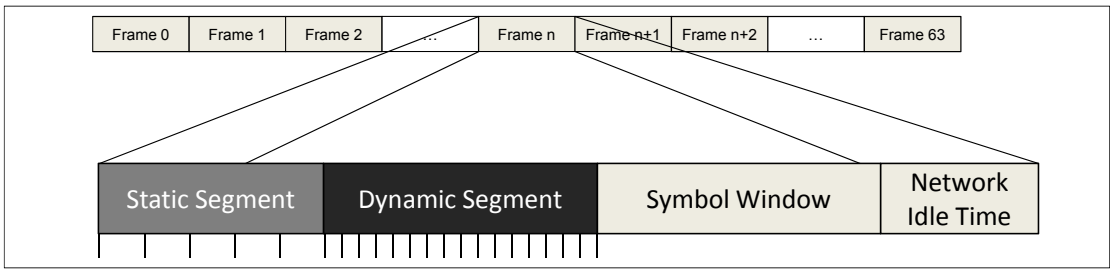


Figure 8.5: FlexRay communication cycles (adapted from [Hei12]).

8.3.2 The Running Example

To initially develop FlexRay protocol components in MontiArc and to later on evaluate the developed FlexRay deployment generator, a running example which simulates the logical architecture of a simplified ACC system has been implemented. An overview of the contained parts of the system is given in Figure 8.6. The ACC is an extension of a regular cruise control. It adjusts the velocity of a car based on the distance to the vehicle in front (`DistanceSensor`) as well as the current velocity (`VelocitySensor`). The `DisplayElement` signals the driver whether the ACC is active. The desired speed and the allowed distance to the vehicle in front is set using the `ControlElement`. The concrete control of the car's velocity is controlled by the `VelocityActuator`.

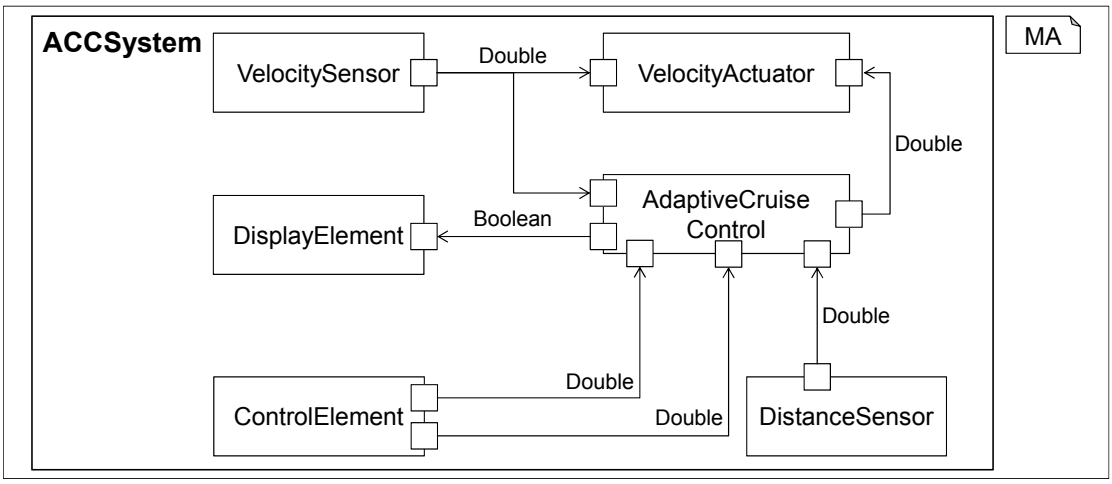


Figure 8.6: Running example of an adaptive cruise control system (adapted from [Rab13]).

8.3.3 Deployment and FlexRay components

To simulate a concrete FlexRay cluster, the running example has been stepwise extended with core and specific FlexRay components. The former represent core FlexRay functionality such as bus drivers or communication controllers without the host interface. The latter are needed to connect the components from the running example to the core FlexRay components.

An exemplary target deployment to a FlexRay cluster is given by component `ACCSystem-Cluster` which is depicted in Figure 8.7. It can be seen that each component from the ACC system is mapped to a node of the FlexRay cluster. The cluster is synchronized using the `TT-D` method. The nodes tagged with the stereotype `«startupNode, syncNode»`, i.e., `node1`, `node4`, as well as `node5`, are startup and sync nodes. The untagged nodes `node2` and `node3` are regular nodes. The `Channels` component represents the abstraction of a passive star [Con10, Section 5.3] to connect the nodes of the cluster with two communication channels. It contains a `Junction` component to forward messages received from a node to all other connected nodes. Additionally, it contains a `Stub` component for each connected node and

communication channel. These stubs realize a configurable bidirectional delay together with a configurable `Disturbance` component. The latter allows to simulate disturbances of a certain communication channel such as bit flips or burst errors. The default delay is configured to 50 nanoseconds which corresponds to a wire length of 25 cm without any disturbance. All depicted node components, the channels, and the main cluster are specific FlexRay components that are needed to perform a FlexRay simulation of the running example. The contained `Delay` components are added to avoid simulation deadlocks within communication cycles. Caused by the Brock- Ackerman anomaly [BA81], such deadlocks occur if a component directly or indirectly communicates with itself and the resulting communication cycle only contains weakly causal components (see Section 3.5.5 on page 73).

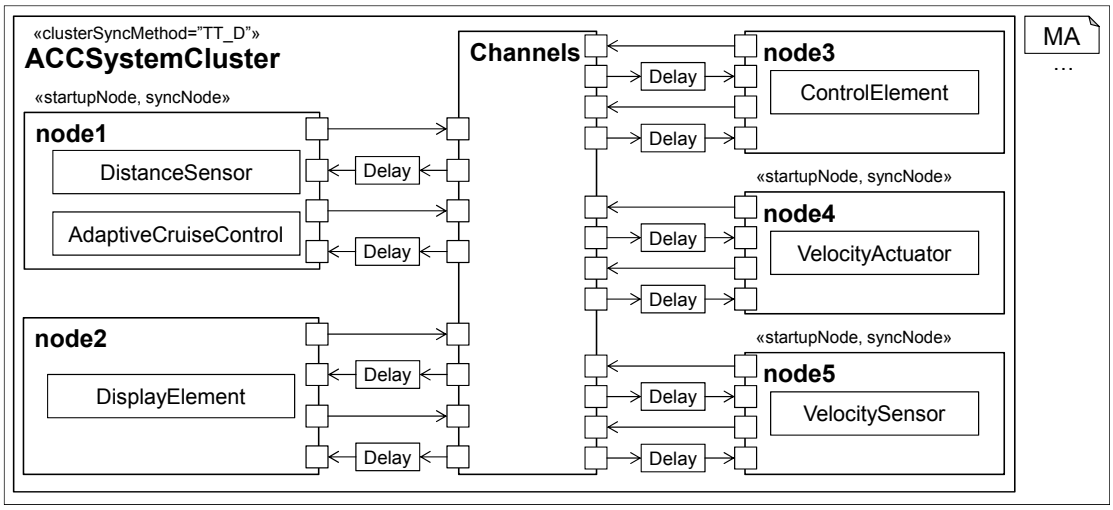


Figure 8.7: Exemplary FlexRay cluster with five nodes which host the components of the cruise control system (adapted from [Rab13]).

The detailed architecture of `node1` is depicted in Figure 8.8. The handcoded components `DistanceSensor` and `AdaptiveCruiseControl` are connected to the FlexRay core components with a specific `HostInterface` which translates application specific messages into general FlexRay messages. The core component `CommunicationController` controls the communication within the FlexRay communication cycles and frames. `BusDriverA` and `BusDriverB`, both FlexRay core components, are responsible to access communication channel A respectively B (see Figure 8.4).

8.3.4 The MontiArc FlexRay Generator

To simplify and automate the tasks to model and implement specific FlexRay components needed to design a FlexRay cluster for MontiArc models, a generator has been developed. Based on a simple language extension with stereotypes (see Section 7.3.1), the generator is able to produce components which define the deployment of a FlexRay cluster similar to Figure 8.7. The available stereotypes to configure the mapping from subcomponents to nodes as well as the con-

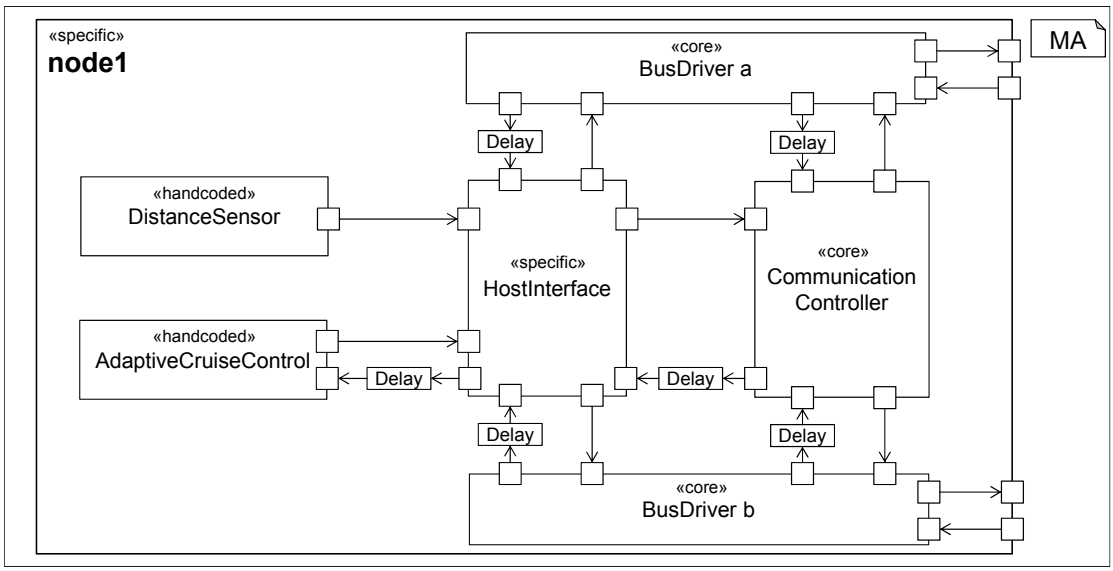


Figure 8.8: Detailed architecture of a FlexRay node component (adapted from [Rab13]).

figuration of the resulting cluster are depicted and explained in Table 8.9. Beside the generated components, the generator additionally creates the behavior implementations of the produced specific atomic components. This way, the generated cluster is ready-to-use.

Stereotype	Default	Description
deployToNode	–	Mandatory for each contained subcomponent. Maps the annotated subcomponent to the node which is given by the stereotype value.
startupNode	false	Optional. The node from the annotated subcomponent is declared to be a startup node. Depending on the cluster synchronisation method, at least one (TT-L) or two (TT-D) startup nodes have to be defined. Each startup node has to be a syncNode, too.
syncNode	false	Optional. The node from the annotated subcomponent is declared to be a sync node.
stubDelay	50	Optional. Configures the delay of the `Stub` component in nanoseconds which connects the associated node within the generated `Channels` component. Please note that simulation time intervals correspond to the timespan of a nanosecond. Thus, the default delay is 50 ticks. The value of the stereotype has to be a positive integer.

Table 8.9 continued on next page

Stereotype	Default	Description
stubDisturbance	NoDistrubance	Optional. Configures the disturbance component of the `Stub` component which connects the associated node within the generated `Channels` component. The value associates the component type which is to be used together with comma separated configuration parameters in the format $parameter\ type$: $parameter\ value$.
clusterSyncMethod	TT-D	Optional. Configures the kind of the generated FlexRay cluster. TT-D as well as TT-L can be used.
attachToChannel	A_B	Optional. Attaches the associated node to the channel(s)) determined by the stereotype value. Allowed values are `A`, `B`, and `A_B`.

Table 8.9: Stereotypes to configure the FlexRay cluster generator.

8.3.5 Conclusion

In the presented diploma thesis [Rab13], a total amount of 25 core and 25 example specific components have been developed. The former are composed of 5 decomposed and 20 atomic components with an average number of 31.0 ports, 108.8 connectors, 15.8 subcomponents, and 121.4 lines of code (without comments). The latter comprise 13 decomposed and 12 atomic components with an average amount of 3.4 ports, 56.7 connectors, 16.7 subcomponents, and 50.6 lines of code. The maximal depth of the component hierarchy is 4 achieved by the exemplary FlexRay cluster. Please note that the most complex component from the running example without any FlexRay components is the `ACCSystem` component with 28 lines of code. Thus, the overall complexity is mostly driven by FlexRay core or example specific FlexRay components. However, the former are provided by a library and the latter can be generated by the developed generator.

Summing up, this case study has shown that it is possible to develop complex protocols and communication infrastructures with MontiArc. The developed generator can be used to generate and compare distinct possible FlexRay cluster configurations for a given MontiArc model. Complex FlexRay components are either available in the developed library or are generated. However, not all FlexRay features given in the specification have been realized and the active star is the only available communication topology. Hence, valuable extensions could be further network topologies specified in [Con10, Chapter 5], e.g., point-to-point connections, a linear passive bus, or an active star network. Further, a configurable communication cycle would be worthwhile. Nevertheless, the overhead to compare and simulate diverse FlexRay cluster configurations has been reduced and an initial base for FlexRay simulations using MontiArc has been developed.

Chapter 9

Language Extension Case Studies

MontiArc and the corresponding tools are designed to be reusable and expendable (see requirement *LRQ3*). Chapter 7 presents a structured method for this purpose. It is based on and refined by experiences made during the development of several languages that extend MontiArc. These extensions have a broad spectrum ranging from the definition of new architectural styles to the application of MontiArc outside the domain of architectural modeling. This chapter initially gives an overview of existing language extensions. Then, three of these extensions are introduced in more detail. First, *AJava* is presented which adds Java elements to MontiArc. Second, *MontiArcAutomaton* is discussed in Section 9.3 which aims at the model-driven development of robots. Third, the process network model and simulation tool ProNet^{sim}, which is based on MontiArc and its simulation framework, is presented in Section 9.4.

9.1 Overview

The following languages and case studies extended MontiArc and the corresponding tool framework:

- *AJava* [HRR10] extends MontiArc with Java methods to directly implement the behavior within atomic components. AJava is presented in more detail in Section 9.2.

- The *Architecture Alignment Checker (AAC)* [MSN11] allows to automatically check the consistency between a Java implementation of a system and its architectural description specified using MontiArc. It provides tool support to automatically detect architectural erosion which is caused by software evolution [PW92]. Therefore, a mapping language has been developed which allows to assign architectural elements to code artifacts such as packages, classes, and methods. By interpreting the architectural model that defines allowed interactions and the given mapping, a rule base is build up which describes the allowed interaction within the target system. The AAC then analyzes the source code of the system to validate adherence of these rules. By integrating the AAC into a continuous development environment, changes that are not in accordance with the architectural model can be detected immediately and disallowed access can be corrected. In this way, architectural conformance can be integrated into an agile development process. The AAC has been evaluated by analyzing the source code base from the SSELab[1] [Her14].

[1] SSELab, `https://sselab.de`.

- The *cloudADL* [NPR13] is part of the clArc workbench[2] which supports the model-based development of cloud applications. It extends MontiArc by adding replicable ports and subcomponents to support dynamic architectures. Consequently, context gates are introduced to model the communication context of replicated components. Further, service ports are integrated into the language to explicitly depict external, not shown communication such as database access of components. The cloudADL extends MontiArc's base-language Architecture Diagram (ArcD) and its symbol table. It further reuses parts of the defined context conditions and transformations.

- *Δ-MontiArc* [HRRS11, HKR+11] provides delta modeling [CHS10] techniques for MontiArc. Delta modeling is a transformational [VG07] software product line (SPL) development approach that allows to describe the difference between two variants in terms of modular deltas. A delta is able to add, replace, modify, and remove model elements. A delta-oriented product line is defined by a core variant which represents a valid product. Further variants are defined by deltas that contain the needed modifications to derive a variant from the core. Since deltas can be also used to describe temporal variability, techniques to evolve delta-oriented SPLs have been proposed in [HRRS12]. Beyond that, Δ-MontiArc is used as the backend of Delta-Simulink [HKM+13] which applies delta modeling to Matlab/Simulink. Additionally, the experiences with Δ-MontiArc and the development of further delta languages led to a constructive method presented in [HHK+13, HHK+15]. This method allows to automatically generate delta-languages for arbitrary textual domain specific languages (DSLs). Δ-MontiArc's grammar extends the MontiArc grammar to inherit the concrete syntax for delta operations. The symbol table is reused within the delta-interpreter which takes a core model and a delta configuration to derive a concrete variant. Subsequently, MontiArc's context conditions are reused to validate generated variants.

- *MontiArcAutomaton* [RRW12, RRW13b, RRW13c, RRW14] extends MontiArc by adding I/O$^{\omega}$ automata [Rum96] to MontiArc components. It is presented in Section 9.3.

- *MontiArc*^{HV} [HRR+11] introduces a hierarchical variability modeling approach to MontiArc. In contrast to Δ-MontiArc, variability within components is restricted to explicit variation points which can be realized by a variant. Variant configurations then map variation points to the concrete realizing variant. Since variation points can be spread across the component hierarchy, selected variants are allowed to explicitly map further variants to other variation points within the component hierarchy. Additional dependencies can be modeled with constraints that determine *requires* and *excludes* relations between variants. Since variation points can be constrained with cardinalities, they enforce a certain semantics which can be automatically validated. For example, since a car is restricted to four doors, a variation point that adds window winder functionality for two additional doors can only be realized once. MontiArc^{HV} comprises two grammars which both extend the MontiArc grammar. First, `HierVarArc` adds the nonterminal `VariationPoint` which implements interface `ArcElement`. In this way, variation points can be used

[2] The clArc Project Page, `http://clarc.cc/`.

within component definitions. Second, the `VariantDefinitionDSL` defines the syntax for variants as well as variant configurations. Therefore, it replaces the header of a component with a specific variant respectively variant configuration header and reuses the component body. Thus, reuse is restricted to inheritance of the MontiArc grammar.

- *MontiArc^{SC}* embeds UML/P [Rum11, Rum12, Sch12] statecharts (SCs) into MontiArc components to constructively model the behavior of atomic components. It has been developed in a small case study conducted to validate the MontiArc extension method described in Chapter 7. MontiArc^{SC} extends MontiArc's language by embedding, realizes the behavior extension point within the symbol table, and derives a new generator by implementing the provided extension points. MontiArc's runtime environment (RTE) is reused without adaption.

- The *MontiSecArc* architecture description language (ADL) focuses on security aspects of distributed reactive systems and is currently under development. Beside analyzing security issues, it aims at enforcing the modeled aspects by generating components which are directly used within the target system. MontiSecArc extends the MontiArc grammar and adds a set of security related model elements respectively refines existing elements accordingly. A few of them are listed in the following:

  - Components can be attached with a *trust level* to analyze whether a component is trustworthy in the current context.
  - Ports can be marked *critical* to define costly or essential functions. This technique is used to identify critical paths within the modeled system.
  - Specialized connectors define *encrypted* or *unencrypted* communication channels. Also a specialized `autoconnect` feature to automatically derive encrypted connections is provided.
  - To share authentication across different components, *identity links* with several *authentication methods* are added.
  - The integration of *user roles* and *access control policies* attached to ports allow to model access rules for ports. Consequently, access to functions or services can be granted to certain users.

  Beside an extension of the MontiArc language, MontiSecArc also reuses its context conditions and extends the provided symbol table infrastructure.

- The *Production Network Simulation (ProNet^{sim})* is a workbench which allows to model, simulate, and assess value chains that are globally distributed within a production network. It is presented in Section 9.4.

9.2 AJava

In software engineering, architectural models serve as crucial development artifacts which define important properties of the system under development. Software architecture brings together

requirements with a structural view of a system which enables a more effective design and program understanding as well as formal analysis [ACN02b].

In classical Model-Driven Development (MDD) resp. Model-Driven Architecture (MDA), it is most often not possible to generate the complete system. Thus, only code frames are derived which need to be adjusted manually. This approach results in a one-shot generation [KM05]. Consequently, after parts of the system have been initially generated and adjusted per hand, further development of the model is complicated since manual adaptions have to be synchronized with re-generated code over and over again. Additionally, evolution often yields to inconsistencies between the architectural model and the generated general purpose language (GPL) code. This either renders the model useless after a certain time or the model needs to be aligned manually with the code [HRR10].

Even in regular non-generative software engineering, where architectural models are not used constructively but for analysis and design, architecture is redundantly present in both: code and model. Since the architecture has been initially developed, it erodes over time. Implicit mappings in the code are often tried to be made more explicit by providing GPL libraries with concepts which represent architectural elements such as JavaBean or FRACTAL components [BCL+06]. Nevertheless, these libraries are restricted to the expressiveness of the GPL. Further, object-oriented classes have an interface that defines which methods are provided by the class. An explicit definition of required methods or functionality is missing.

These problems can be encountered in two different ways that merge architectural modeling and programming. First, architectural elements can be added to a GPL. Second, GPL elements can be added to an ADL. The former is related to the aforementioned library approaches or in a more consequent way realized by ArchJava [ACN02b, ACN02a] and Java/A [BHH+06] (see Section 2.3.4). The latter approach embeds GPL code into first-level architectural elements. For example, ComponentJ [SSP08] introduces components with embedded Java methods, provided, and required interfaces. Similar to ArchJava, connecting these interfaces corresponds to a transparent delegation of synchronous method calls between objects. In contrast, AJava [HRR10] adds Java elements to MontiArc components to design an asynchronous, event-driven architectural programming language (APL).

9.2.1 Example

An exemplary AJava architecture which comprises the components of a coffee machine is depicted in Figure 9.1. It internally defines an enumeration `CoffeeType` that contains valid coffee choices which are served by the machine. Its main component is the `CoffeeProcessingUnit` which receives the filling quantities of coffee- and espresso beans from the corresponding `BeanSensor` subcomponents `espressoBs` and `coffeeBs`. It is connected to an external milk frother via the ports `milkEmpty`, to be informed whether the milk is empty, and port `milkAmount`, to request a certain amount of milk. Finally, it contains a display subcomponent to communicate with the user that enters his coffee choice via port `selection`.

An excerpt of the `CoffeeProcessingUnit` AJava implementation is depicted in Listing 9.2. Its interface is given by the ports in ll. 2 – 8. It internally stores the availability of milk in a private field which is declared in l. 10. Its reaction to events received on port `milkEmpty` is implemented in the Java method `onMilkEmptyReceived` (cf. ll. 12 – 20). If the milk

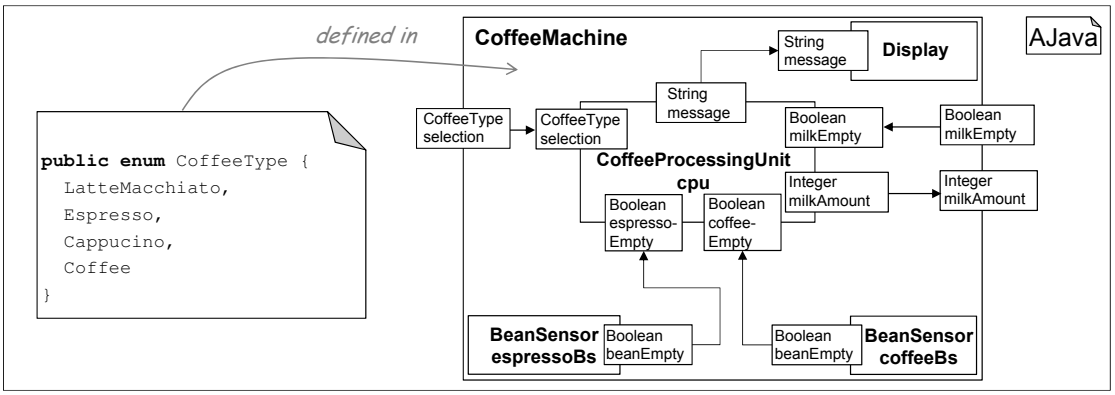


Figure 9.1: Architectural model of a coffee machine (according to [HRR10]).

```
1  component CoffeeProcessingUnit {                                    AJava
2    port                                                               ...
3      in CoffeeType selection,
4      in Boolean espressoEmpty,
5      in Boolean coffeeEmpty,
6      in Boolean milkEmpty,
7      out Integer milkAmount,
8      out String message;
9
10   private boolean milkAvailable;
11
12   public void onMilkEmptyReceived(Boolean milkEmpty) {
13     if (milkEmpty) {
14       this.message.send("Sorry, no milk today.");
15     }
16     else {
17       this.message.send("Got milk!");
18     }
19     this.milkAvailable = !milkEmpty;
20   }
21 }
```

Listing 9.2: Implementation of the `CoffeProcessingUnit` in AJava syntax (according to [HRR10]).

tank is empty, it sends the message `"Sorry, no milk today."` (cf. l. 14) via port `message`. If the tank got filled up, it sends `"Got milk!"` via the same port (cf. l. 17). Finally, it stores the state of the milk tank in the local field `milkAvailable` to be able to properly react to selection requests. The shown implementation obeys two implications. First, for each incoming port an event-processing method has to be implemented that handles the corresponding events. In the example, this is demonstrated for port `milkEmpty` with the corresponding `onMilkEmptyReceived` method. Further event-handling methods are omitted for reasons

of space. Second, the outgoing ports can be accessed like fields of a class and provide methods to emit messages. This is demonstrated with the `send` method of port `message` (cf. ll. 14, 17). Available methods are defined by the API of the underlying RTE and generated code.

9.2.2 Language and Tool Extensions

AJava builds up on MontiArc by extending its syntax, symbol table, generators, and RTE. It adds a selected set of Java elements to the MontiArc language: local class definitions, methods, constructors, as well as annotations.

Language Extension: The involved MontiCore grammars that define the syntax of AJava are depicted in Figure 9.3. The `AJava` grammar extends the `MontiArc` grammar and inherits all language elements. It also adds an external nonterminal and a simple nonterminal. The defining production of the latter implements the interface `ArcElement` and its right-hand side (RHS) contains a reference to the created external. Since it implements the `ArcElement` interface, it can be used within the body of a component definition (see Listing 3.20 on page 54). Details about the external embedding mechanism provided by MontiCore are given in [KRV10, Kra10].

Since only a subset of Java elements should be available in components, the regular `JavaDSL` [FM07] is extended by grammar `AJava_JavaDSL`. A production of the latter defines the nonterminal `AJavaEmbedments` which bundles the Java elements that should be added to AJava as alternatives within its RHS. Finally, to use this nonterminal within AJava, it is embedded into the created external nonterminal. In this way, only the Java elements bundled within the RHS of production `AJavaEmbedments` are available in AJava.

Beside the syntax, AJava also extends MontiArc's symbol table. For this, an AJava method entry, corresponding resolvers, and qualifiers are implemented to represent methods and constructors of a component. MontiArc itself already provides field entries to represent configuration parameters which are reused to represent fields within a component (see Figure 5.8 on page 135). Additionally, adapters are created to convert Java field entries into AJava field entries and Java methods into AJava methods. Java entries are automatically build up by the Java symbol table when processing the embedded elements. Further, the component and subcomponent entries are refined within a subclass that also aggregates the newly created entries. Adapters and resolvers are added to interpret a port entry as a field entry with the type `IOutPort<T>`, whereby `T` corresponds to the type of the port.

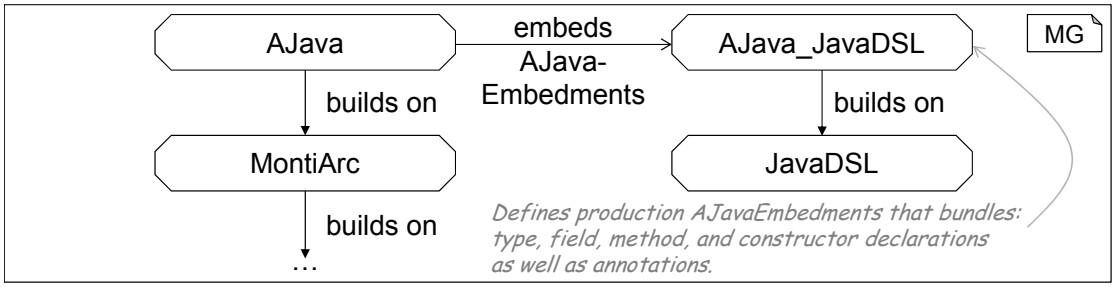


Figure 9.3: MontiCore Grammar hierarchy and embeddings for the AJava language.

Model Processing Extension: Three context conditions have been added to enforce the aforementioned implications. The first validates whether for each incoming port of an atomic component a corresponding event-handling method is implemented. The second ensures that constructors are implemented correctly and have a parameter for each configuration parameter of a configurable component. The third checks that no methods are defined within decomposed components. Additionally, four transformations are added that qualify Java types within the AJava abstract syntax tree (AST) of newly added Java model elements.

Simulation Extension: AJava components are compiled against the regular MontiArc RTE which has been slightly extended to enable physical distribution of components. The added classes and interfaces as well as their relation to elements from MontiArc's RTE, these elements have a gray background color, are depicted in Figure 9.4. Interface `IInTCPPort` extends the interface of an incoming simulation port and adds a method to start listening for messages on a given TCP port. Internally, it starts the associated `Runnable InPortTCPServer` which is executed in an own thread and is responsible for receiving messages via the configured TCP port. Received messages are delegated to the `accept()` method of the corresponding `IInTCPPort` (see Figure 4.10 on page 94).

Message sending is realized by interface `IOutTcpPort` which extends the interface of an outgoing simulation port. It adds a method to register a receiver which is accessible by the given IP address and TCP port. It internally creates and starts an `OutPortTCPServer Runnable` in a dedicated thread. Messages, which are send using the `send()` method of an `IOutTcpPort`, are delegated to all associated servers that perform the TCP transmission.

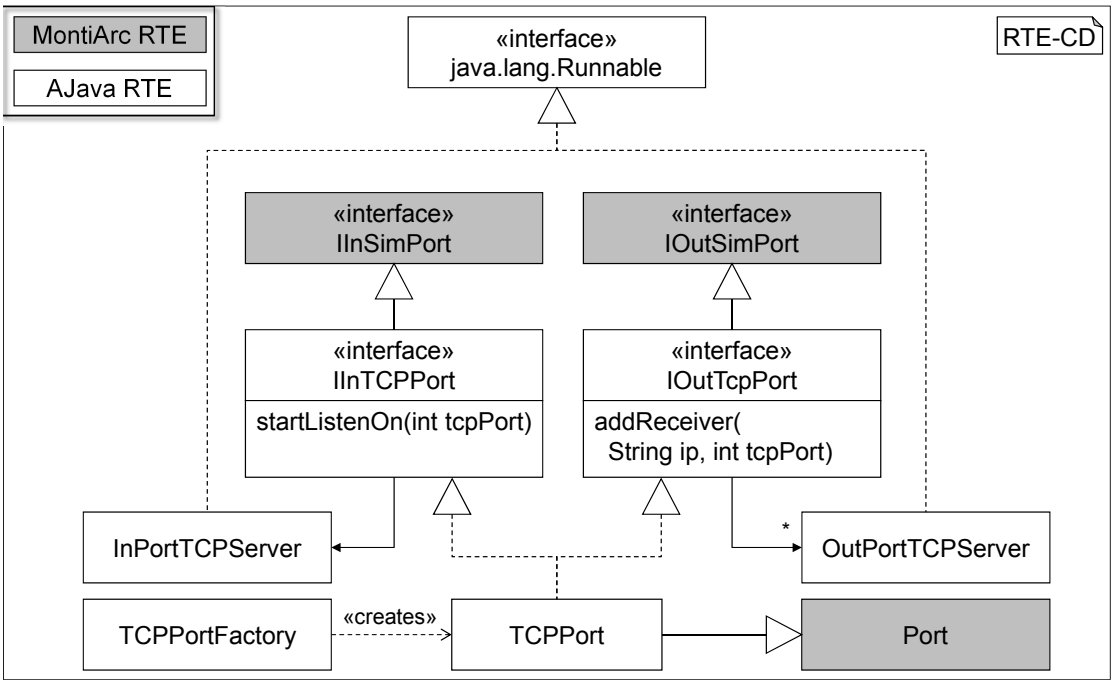


Figure 9.4: AJava RTE extensions of the MontiArc RTE.

A concrete implementation of these interfaces is given by class `TCPPort`. It inherits the implementation from the regular simulation port. Only the aforementioned message delegation to and from the servers is realized in the `TCPPort` class. Instances are created by the `TCPPort-Factory` which can be passed to any simulation scheduler. TCP connections are created by calling the `startListenOn` respectively the `addReceiver` methods of system components. Please note that a `TCPPort` acts like a regular simulation port if no servers are registered. In this way, only the outer ports of a system component communicate via TCP. Ports of contained connected subcomponents communicate like regular MontiArc components.

Since the presented RTE extensions are integrated into the simulation by a factory and the activation is performed by the configuration of system components, the RTE extensions intrinsically do not require any extensions of the generator. However, an adaption of the generator is needed, since atomic AJava components, in contrast to atomic MontiArc components, directly contain their implementation. Consequently, the generator has to produce concrete instead of abstract classes for an atomic AJava component. Further, it has to integrate the embedded Java elements into the generated class. Therefore, two new configuration templates are created, four templates replace existing templates (component, component body, constructor, and factory) and a template which integrates the embedded Java code is added. Also four new template calculators are added. Consequently, a distributed AJava system corresponds to several MontiArc simulations that are interconnected via TCP remote ports.

9.2.3 Conclusion

The presented language AJava proposes a feasible way to integrate concrete behavior implementations into the ADL MontiArc which renders a mapping between architecture and code superfluous [HRR10]. Since MontiArc and its tools are designed for extension, the effort to realize AJava has been minimized. The experiences made during the development of AJava have been directly integrated into the MontiArc framework to further ease reuse of the language and the corresponding tools. For example, an analysis of the AJava generator lead to further provided extension points. These experiences also influenced the MontiArc extension method presented in Chapter 7 and the development of AJava serves as an evaluation of this method.

Since AJava has been developed based on an older MontiArc version, some extensions would not be needed if using the current version. Now, creating a new component and constructor template is superfluous since resulting class names and modifiers of generated classes are determined by the configuration within the main template. Further, it is easier to add additional templates which are called to generate content into the resulting class body.

AJava serves as both, a case study for MontiArc extensions as well as a case study for architectural programming. The initially developed compiler can be improved, e.g., by adding a deployment generator which synthesizes the configuration of system components or by adding IDE support. Nevertheless, it has been shown that non-trivial extensions of the MontiArc language can be developed that reuse large parts of the provided MontiArc tools. In this way the development effort for such languages is reduced.

9.3 MontiArcAutomaton

MontiArcAutomaton [RRW12, RRW13b, RRW13c, RRW14] is an extension of MontiArc which targets at the development of reusable components for the robotics domain. Even simple robotics applications are inherently complex since they consist of multiple distributed components which have to be integrated with each other. Nowadays, robotics applications are still highly experimental, monolithic, and hardly adaptable to different platforms [Mos09, SSL11]. Even though reuse in robotics is heavily pursued, it mostly is performed on a binary component level [BBC+07] which still renders the application to be suited for a few target platforms only.

MontiArcAutomaton tries to cope with these problems by integrating MontiArc with I/O$^{\omega}$ automata [Rum96]. This way, robots are structured using MontiArc's hierarchical decomposition mechanism, logical as well as physical independent components are developed separately. The control logic is constructively modeled and thus realized with language-agnostic automata. A generative approach enables reuse on different target platforms by transforming MontiArcAutomaton models into concrete GPL code which is directly deployed on the target.

Beside the generation of target code, MontiArcAutomaton models are used for analysis. This comprises type system checks inherited from MontiArc, context condition checks, as well as formal model checking. An approach to validate the refinement of MontiArcAutomaton components using the model checker Mona [EKM98] is outlined in [RRW13b] and presented in [Rin14]. The MontiArcAutomaton framework has been evaluated in [RRW13a].

9.3.1 Example

A brief example for MontiArcAutomaton models, which has been introduced in [RRW13b], is depicted in Figure 9.5. A simultaneous localization and mapping (SLAM) bot is built to discover and map unknown areas. Due to missing position sensors, such as GPS, the robots have to estimate their position based on their starting position. Thus, the more time has passed, the more increases the difference between the estimated and the real position. To reduce this difference and increase the quality of the localisation, the extended SLAM bot also communicates with other bots for a cooperative discovery of the environment. The SLAM bots are built with a NXT Lego Brick and consists of the following physical components: a `TouchSensor` to detect obstacles, two `Motor` components `mRight` and `mLeft` which realize the powertrain, and a `Bluetooth` component to interact with other bots. The control logic is given by a `BumpControl` component responsible for the steering, a `Timer` to realize timeouts and the explored map is continuously build up by component `MapBuilder`. Please note that physical components are parametrized with additional deployment information. In this way the `TouchSensor` is connected to `SensorPort.S1` of the NXT Brick.

The logic that controls the autonomous movement is pretty simple. The SLAMRobot should drive in a straight line until it hits an obstacle with its touch sensor. It will then back off, turn a bit, and continue exploring the area by driving straight forward. The controls of the SLAM bot, which realize this driving logic, have been designed in a hardware-independent manner. The realization of component `BumpControl` is given in Listing 9.6. Just like in the regular MontiArc language, the component interface is defined by a set of incoming and outgoing ports (cf. ll. 2 – 7). The behavior is declared by the embedded automaton (cf. ll. 9 – 21) that consists

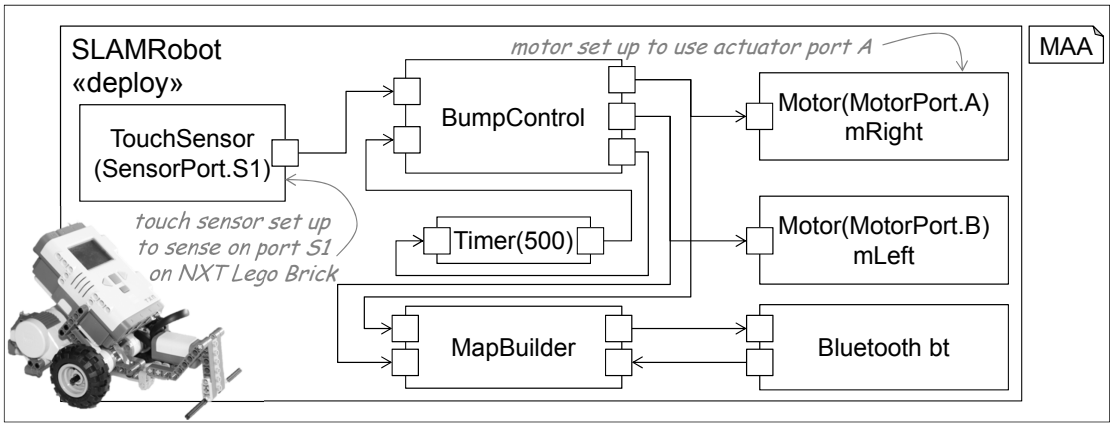


Figure 9.5: A SLAMRobot constructed with an NXT Lego Brick. The architecture and the behavior of SLAMRobots is modeled with MontiArcAutomaton (according to [RRW13b]).

of four states: `idle`, `driving`, `backing`, and `turning` (cf. l. 10) . It starts in the initial state `idle` by emitting the `STOP` command as initial values to both connected motors via the ports `rMot` and `lMot` (cf. l. 11). The contained transitions (cf. ll. 13 – 19) induce the automata to switch from source to target state and to perform an action when they are triggered. Defined by the transition given in ll. 13f, the automata respectively the `BumpControl` switches from state `idle` to `driving` if `true` is received on port `bump`. This events corresponds to an initial activation of the touch sensor after starting the robot. Afterwards, the command `FORWARD` is emitted on both ports `rMot` and `lMot` to drive straight forward.

9.3.2 Language Extensions

To be able to define the behavior of MontiArc components with I/O$^{\omega}$ automata, MontiArc has been extended to the initial MontiArcAutomaton version presented in [RRW12]. Figure 9.7 a) depicts that the `MontiArcAutomaton` grammar directly builds on the `MontiArc` grammar. It extends its base-language by adding two productions which implement the `ArcElement` interface. Consequently, they can be used directly within the body of a component (see Listing 3.20 on page 54). The MontiArcAutomaton grammar is provided in [Rin14, Appendix J].

First, I/O$^{\omega}$ automata are added that itself contain the following model elements. States are used to define the state-space of a component. Transitions are used to define the triggers as well as the resulting actions for changes of state.

Second, local variables are added which allow to store values during the action of a transition. Since these variables can be also referenced within the trigger condition of a transition, they implicitly extend the state-space of the automaton. This technique basically counters the explosion of states and allows to easily implement loops.

Since new language elements are added, also the symbol table is extended. Therefore, the component entry class (Figure 5.8 on page 135) is extended to additionally contain automata

MAA

```
component BumpControl {
  port
    in Boolean bump,
    in TimerSignal ts,
    out TimerCmd tc,
    out MotorCmd rMot,
    out MotorCmd lMot;

  automaton {
    state idle, driving, backing, turning;
    initial idle / {rMot = STOP, lMot = STOP};

    idle -> driving {bump = true} /
        {rMot = FORWARD, lMot = FORWARD};
    driving -> backing {bump = true} /
        {rMot = BACKWARD, lMot = BACKWARD, tc = DOUBLE_DELAY};
    backing -> turning {ts = ALERT} /
        {rMot = FORWARD, tc = DELAY};
    turning -> driving {ts = ALERT} / {lMot = FORWARD};
  }
}
```

Listing 9.6: The `BumpControl` component with an embedded I/O$^{\omega}$ automata (according to [Rin14]).

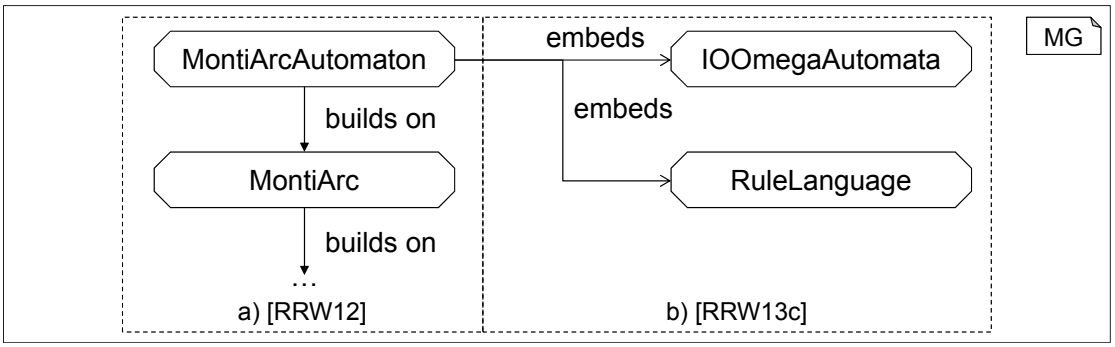


Figure 9.7: MontiCore Grammar hierarchy for a) the initial MontiArcAutomaton version that uses inheritance and b) the current MontiArcAutomaton version that uses embedding to extend MontiArc with I/O$^{\omega}$ automata.

entries and entries that represent variables. With these extensions, MontiArcAutomata is able to reuse most parts of MontiArc's symbol table and a selected set of context conditions. It provides its own leJOS [3] based RTE and thus is not able to reuse (parts of) the MontiArc code generator presented in Section 5.4.

With this initial language setup, one is already able to design robots in a target-language-

[3] LEJOS - Java for LEGO Mindstorms, `http://www.lejos.org/`

agnostic way with MontiArcAutomaton. However, it determines several drawbacks. Using language extension to add automata to MontiArc components results in hard-wired dependencies between automaton-defining productions and MontiArc. Particularly, such automata always have to be defined within a component and can not be used stand-alone as top-level compilation units.

To encounter these problems, the MontiArcAutomaton language has been revised as depicted in part b) of Figure 9.7. Basically, the productions which define I/O$^{\omega}$ automaton nonterminals are extracted from grammar `MontiArcAutomaton` into a dedicated `IOOmegaAutomata` grammar. The top-level automaton rule is then embedded into the external nonterminal `BehaviorEmbedding` which is provided by MontiArc (see Listing 3.25 on page 56). As discussed in [RRW13c], additionally a `RuleLanguage` is embedded into the same extension point that allows to define component behavior in an input/output relation given in a table-like syntax.

As already mentioned, MontiArcAutomaton does not reuse MontiArc's RTE and the corresponding generators. Instead it provides a set of RTE/generator combinations itself. These are:

- A leJOS RTE and corresponding generators which allow to deploy MontiArcAutomaton models to Lego NXT Bricks is presented in [RRW12, Rin14].
- A Python based RTE and generators with direct ROS[4] integration.
- A generator for Mona verifications to enable formal reasoning about automata refinement is discussed in [RRW13b, Rin14].
- For visualisation, an EMF core generator is provided [RRW13b].

9.3.3 Conclusion

The (ongoing) development of the MontiArcAutomaton language has shown that reuse and extension of the MontiArc language fronted is possible and worthwhile. Beside the syntax, MontiArcAutomaton additionally reuses and extends MontiArc's symbol table and a selected set of context conditions. The generator and RTE are not reused.

The experiences made during the development of MontiArcAutomaton directly influenced the language extension method for MontiArc (see Chapter 7). Beside a practical use case and requirement providing stakeholder, MontiArcAutomaton has been the main influence to a) add a behavior definition extension point to the MontiArc grammar and b) to figure out techniques and methods for extensions within MontiCore's symbol table framework. Actually, the external nonterminal which is provided to embed arbitrary behavior definition languages leads to the concept of a generic extension point in MontiArc's symbol table (see Section 7.3.2).

[4]The Robot Operating System, `http://www.ros.org/`

9.4 Process Network Simulation

The Process Network Simulation (ProNet^{sim})[5] aims at modeling and simulation of distributed value chains within a global production network. Rodrigue et al. [RCS13] define a value chain as a "[...] functionally integrated network of production, trade and service activities that covers all the stages in a supply chain, from the transformation of raw materials, through intermediate manufacturing stages, to the delivery of a finished good to a market". Additionally, they claim that each of these activities corresponds to a node which transforms its inputs to one or many outputs with an added value. In the following, these nodes are called process steps. Each process step can be assigned to a different location which leads to a value chain that is distributed across several locations. On the one hand, production costs can be reduced by this course of action. On the other hand, transport costs and total production time are increased. Since the total added value and processing time of such a value chain interrelate with much more parameters, an automatic simulation and comparison of different scenarios is helpful to assess a concrete scenario.

Since the graphical representation of value chains is pretty similar to the graphical syntax of MontiArc, values chains have been initially modeled as plain MontiArc models regarding a simple mapping:

- Value chains correspond to decomposed and production steps to atomic components.
- Subcomponents either represent another value chain or a process step.
- Transports between nodes of a value chain are represented by connectors.
- Commodities, i.e., raw materials or (partially) finished goods, are described by data types.
- Input- and output ports correspond to the input and the resulting output of a process step or value chain.
- Further variable parameters that together define one or many scenarios are defined within a spreadsheet.

In this way, the simulation of value chains has already been possible. Nevertheless, MontiArc's syntax seem to be less intuitive for process network domain experts. Also, to adjust the semantics of the MontiArc simulation to the semantics of a value chain, some intermediate components have to be introduced that blurred the intended value chain. This semantic difference is discussed in Section 9.4.2.

9.4.1 Example

An exemplary value chain `PipeProduction` for the industrial manufacturing of pipes is depicted in Figure 9.8. Its input is given by `Iron` and `Zinc` which are casted to a `PipeWorkpiece` in the `Casting` process step. The workpiece is then transported to the `Assembly` process step. Further, `Plastic` and `Gum` are processed to `SealingMaterial` by the `SealingMaterialProduction`. The `SealingMaterial` is passed to the `SealProduc-`

[5] ProNet^{sim} has been funded by a RWTH University Dean's Seed Fund in 2011. It has been developed in cooperation with the Production Engineering chair, Laboratory for Machine Tools and Production Engineering (WZL), RWTH Aachen University. We especially thank our research assistants Juha Veikko Lauttamus, Sebastian Roidl, and Minh Quan Tran for large parts of the implementation as well as Daniel Kupke (WZL) for the constructive feedback and fruitful discussions.

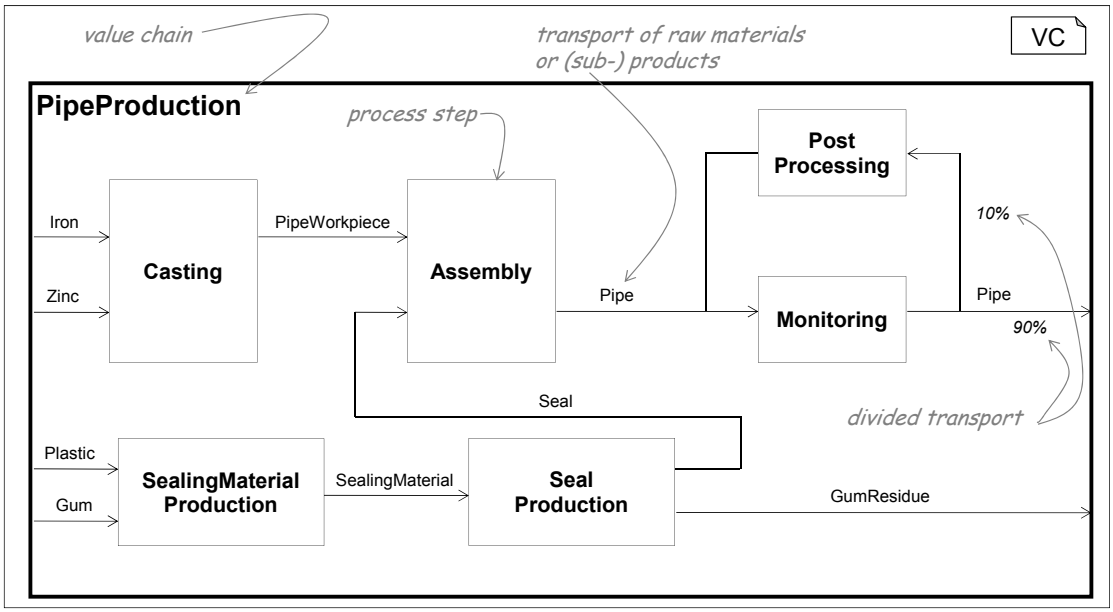


Figure 9.8: Exemplary value chain that models the dependencies between different process steps to manufacture pipes.

tion which produces Seals. The Assembly combines the seals with a workpiece to a Pipe. Since commodities cannot be copied, a transport with more targets has the semantics of a choice. Thus, such connectors have to be tagged with percent values that define the probability, that a commodity emitted by a sender is transported to a certain receiver. If these values are omitted, the commodities are evenly spread to all receivers. Consequently, after monitoring the pipes, 10 % of them have to be post processed and are sent back to the Monitoring process step. 90 % are expected to pass the monitoring immediately. Summing up, the depicted value chain produces pipes as main and gum residue as side product.

To be able to simulate the modeled value chain, at least one scenario has to be configured. Therefore, several properties have to be set within a predefined spreadsheet. Figure 9.9 depicts an excerpt of four scenarios defined for value chain PipeProduction. It has been created by performing the following activities:

1. Define locations, their energy costs, and the hour rates per wage group. A location is referenced by its Code. In the example, two locations are defined: Germany and Poland. Please note that a location must not necessarily correspond to a country. It is also possible to define several locations within the same country.
2. Calculate the distance between all locations. Since street routing may not be symmetric, always both directions between two locations have to be given. In the example, the transport distance from Germany to Poland amounts to 780 km, while the transport distance to other direction amounts to 760 km.
3. Define transport costs per unit and km for all commodities and how many units of a commodity can be transported together (not shown in the example). The last property assumes

Location	Code	energy costs (€ / kWh)	wage group I (€ / h)	wage group II (€ / h)
Germany	D	0.18	40	60
Poland	PI	0.11	8	10

VCCfg
…

Distance (km)	D	PI
D	0	780
PI	760	0

Commodity	Transport Costs (€ / Unit per km)	…
Iron	0.01	…
Pipe	0.10	…

Machine ID	Energy Consumption (kWh)	Machine-Hour Rate (€ / h)	Location	Name
D.Casting11	20	60	D	Casting11
PI.Casting9	35	65	PL	Casting9
PI.WorkStat01	10	80	PL	WorkStat01
D.WorkStat0815	5	30	D	WorkStat0815

Process Step	Machine ID	Length (min)	Effort in wage group I (min)	Effort in wage group II (min)
Casting	D.Casting11	100	30	0
Casting	PI.Casting9	110	35	0
Assembly	D.WorkStat0815	5	5	0
Assembly	PI.WorkStat01	5	5	0

Figure 9.9: Scenario definition for value chain `PipeProduction` within spreadsheets.

a standardized transport container.

4. Model the machines that are available at the defined locations. For convenience, the ID of a machine consists of the corresponding location code and the name of the machine. Further, a machine consumes a certain amount of energy. The machine-hour rate defines the costs per hour.
5. Design a production plan by defining the amount of input commodities together with the costs per unit (not shown in the example).
6. By mapping process steps to machines in association with the needed machine time and effort for each wage group, scenarios are defined. If more than one mapping for a process step is given, the combination of all valid mappings result in multiple scenarios. Assuming that each non-shown process step of the example is uniquely mapped to a single machine, the configuration in Figure 9.9 produces exactly four scenarios. Please note that more than one process step can be mapped to a single machine. In that case, the machine can only be used by a single process step at each point in time.

9.4.2 Discuss Language Extension

As already mentioned, the semantic of a ProNet^{sim} simulation differs from the semantic of a MontiArc simulation. This especially manifests in the following items:

- *Commodities cannot be broadcasted:* In MontiArc, messages are simply transmitted to all connected receivers of a sender. This is possible since data messages can simply be copied. Physical commodities cannot be broadcasted and are transported to a single receiver only. Thus, a one-to-many connection has the semantics of a broadcast in MontiArc and the

semantics of a choice in ProNet^{sim}.

- *Multiple senders:* In MontiArc, the sender of a message always has to be unique while a process step can receive the same commodity type from different senders. Consequently, many-to-one connections are forbidden in MontiArc while they are allowed in ProNet^{sim}.
- *Transport time:* According to the underlying FOCUS framework, transmitting a message is instantaneous. In contrast, transport of commodities always takes time according to the defined configuration.
- *Locking resources:* Since MontiArc aims at modeling logical functions, the concretely used hardware resources are out of scope and are not regarded within a simulation. In contrast, process steps are mapped to physical machines and need exclusive access. On that account, resources need to be locked while processing input commodities to prevent access from further process steps.

Following Karsai et al. [KKP$^+$09], elements with different meaning respectively semantics should differ in their representation, too. Thus, MontiArc can still be used as input for the code generator, but a domain-specific frontend is needed which matches the semantics of value chains and additionally is intuitive for process network experts.

Consequently, a ProNet^{sim} DSL is developed which allows to model value chains and process steps. The essence of the developed grammar is depicted in Listing C.4 on page 312. Each compilation unit defines either a value chain or a process step, has a name, and contains arbitrary many process definition elements. The provided elements are:

- **Value chain nodes** either instantiate another value chain or another process step. The type of the instantiated node is followed by an optional name. The name has to be used if more than one node with the same type is instantiated. Finally, a transport directly connects the node with further nodes or output commodity channels.
- **Commodity channels** define the input or output of a certain commodity type. The optional amount defines how many units of a certain commodity are needed to produce output commodities. In a value chain, a transport directly connects input channels with the receiving contained nodes.

The targets of a transport can be annotated with a probability value. If one source (either a commodity channel or value chain node) is connected to more than one receiver, the sum of the target probabilities has to be 100.

To reuse MontiArc's generator and its RTE, ProNet^{sim} models have to be transformed to MontiArc models while obtaining the intended semantics. This is performed in the following steps:

- ProNet^{sim} models are pretty printed into MontiArc syntax. Model elements or attributes that do not have a MontiArc counterpart are represented by stereotypes. Ports that represent a commodity channel with annotated amount are tagged with an `≪amount≫` stereotype with an assigned value. Connectors that represent transports with multiple targets are annotated with a `≪probability≫` stereotype with a comma-separated value for each receiver.
- To prevent simulation deadlocks (see Section 3.5.5), a delay component is added to all feedback loops.
- To support many-to-one connections, merge components are added which forward re-

ceived messages respectively commodities from all incoming ports to the single outgoing port.

To adjust the semantics of a value chain simulation accordingly, also MontiArc's RTE is extended as depicted in Figure 9.10. Please note that interfaces with a gray background color belong to MontiArc's RTE. Process steps in the simulation are represented by interface `IProcessStep` which extends the regular component interface. To gain access to the concretely configured costs and energy consumption, a process step is associated with a concrete machine (`IMachine`) which belongs to a location (`ILocation`). A process step collects received input commodities until the needed amount of all commodity types is reached. Then it registers at the assigned machine to acquire the next free production slot using method `register()`. If the machine is ready, it is blocked while the process step is producing its output commodities that are represented by the interface `ICommodity`. When the process step finished the production, the associated machine is released using method `release()`. Also special `IProNetPorts` are added which realize the functionality to send commodities to one receiver exclusively instead of broadcasting a commodity to all connected receivers. To be able to properly compute the transportation costs to the receiver, the sender of a commodity is passed as parameter to the `send()` method. Added receivers will receive produced commodities based on their associated probability (`prob`) passed to method `addReceiver(rec, prob)`.

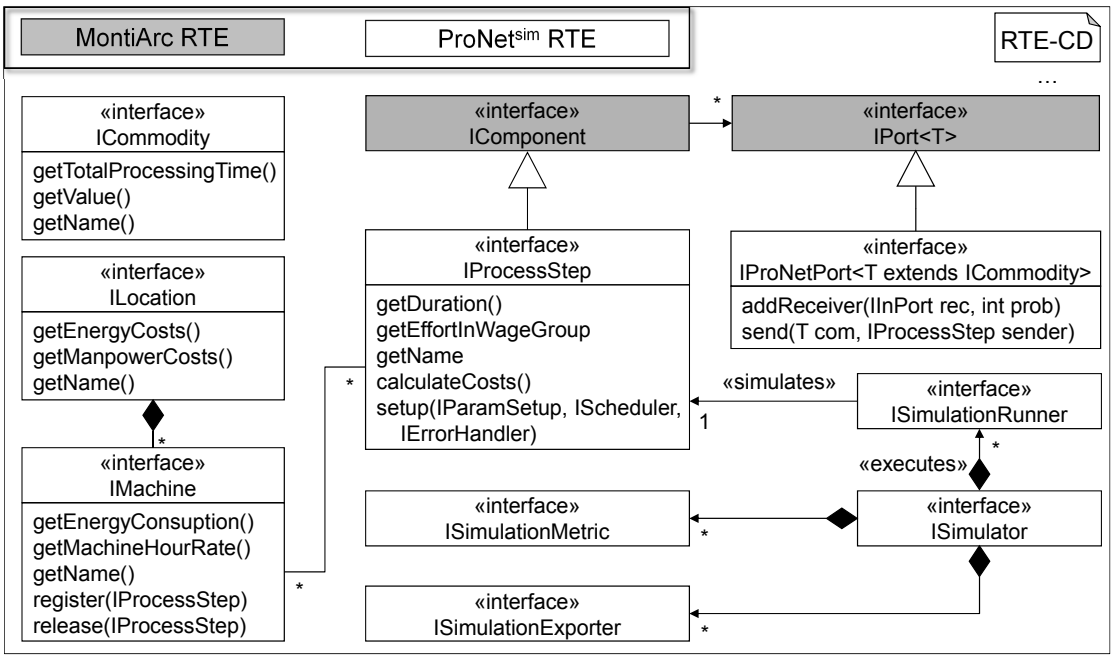


Figure 9.10: ProNet^{sim} extensions of the MontiArc RTE.

An additional generator converts the configuration given in the spreadsheet into configuration classes which implement interface `IParamSetup`. For each contained scenario, a simulation runner is generated that implements `ISimulationRunner`. It sets up the associated process step with the parameter configuration and is able to simulate the process step. These runners

are executed by a central simulator (`ISimulator`) in a dedicated thread which also collects the results of finished simulations. Consequently, multiple scenarios are simulated in parallel. When all simulations have delivered their results, the simulator triggers registered metrics (`ISimulationMetric`) to analyze the results. Exemplary metrics are: the *lowest average processing time*, the *lowest total processing time*, and the *lowest total costs* metrics. The results of the simulation and metric executions are finally exported using the registered `ISimulationExporter`. Since a preconfigured simulator is generated too, no manual programmatic configuration is needed.

Summing up, ProNet^{sim} reuses MontiArc's language to be able to reuse the corresponding generator which is designed for the MontiArc AST. In doing so, also the symbol table, the transformations, and a selected set of context conditions are indirectly reused. Regarding the derived ProNet^{sim}-MontiArc generator, only a small amount of original MontiArc templates and template operators are instantiated within the ProNet^{sim} generator. The complete generator has been initially copied and adjusted to generate according to the ProNet^{sim} RTE. Afterwards, 13 unmodified templates have been removed and the original templates have been reintegrated from the classpath.

9.4.3 Conclusion

With the presented ProNet^{sim} workbench one is able to model value chains with an intuitive domain-specific syntax. Flexible configurations and scenario definitions of the modeled value chain are defined in an Excel spreadsheet template which has shown to be very comfortable for domain experts. The generated simulation can be executed out of the box without hand-coding a single line of code. ProNet^{sim} can be easily extended with further metrics or report generators which can be seamlessly integrated into the RTE architecture.

Nevertheless, further improvements can still be integrated into ProNet^{sim}. More fine-grained values and configurable parameters lead to more reliable and detailed simulation results. For example, different storage costs, import duties for transports across borders, a fine-grained differentiation between transport-, storage-, as well as production durations, or changeover costs for machines used by different process steps are possible. The latter also motivates the integration of sophisticated, configurable, and interchangeable machine scheduling strategies.

Regarding reuse and extension of MontiArc, ProNet^{sim} also demonstrates that it is possible to reuse MontiArc's language, generators and RTE to extend them with an adjusted (runtime) semantics. Since ProNet^{sim} has been developed based on a very early MontiArc version, which comprises a generator that has not been designed for extension, only a few templates and calculators could be reused via the classpath. Although, one can argue that copy and paste is a form of reuse, it is definitely not the best and cleanest form. However, by adapting ProNet^{sim} to the most current MontiArc version, duplication of templates could be completely omitted by using well defined extension points of the generator.

Chapter 10

Discussion and Conclusion

The presented architecture description language (ADL) MontiArc has been designed based on a detailed set of language and simulation specific requirements which have been derived from the posed research questions (see Chapter 2). These requirements are important for an extendable ADL (*RQ2*) which is also used as an architectural programming language (APL) for rapid prototyping of such systems (*RQ4*). The developed ADL MontiArc has a core architectural style which corresponds to the FOCUS modeling framework [BS01]. Since the core language only provides mandatory ADL modeling elements, i.e., components, connectors, and architectural configuration (see [MT00]), it is easy to learn (*RQ1*). Together with the underlying FOCUS semantic framework, it provides enough information to simulate MontiArc models which can be composed from user-defined and library components (*RQ3*). The corresponding simulation runtime environment (RTE) and FOCUS scheduling strategies are presented in Chapter 4. The technical realization of the language and the corresponding tools are defined in Chapter 5.

The structured method presented in Chapter 7 is suitable to adjust the language and the architectural style. It allows to add more detailed model elements to the language, to extend and refine existing elements, or to integrate concrete behavior definitions into component definitions. In this way, MontiArc can be adjusted to specific needs of the target domain.

The reminder of this chapter starts with a detailed discussion whether MontiArc fulfills the defined requirements for architectural modeling and simulation. Finally, the thesis is concluded in Section 10.3.

10.1 Requirements for Architectural Modeling

The language related requirements given in Section 2.1 are derived from the posed research questions *RQ1*, *RQ2*, and *RQ3*. The following discussion clarifies how these requirements are met by the developed MontiArc ADL.

10.1.1 *LRQ1*: Architectural Style

MontiArc is developed as a basic ADL with a single core architectural style. It provides the mandatory architectural modeling features defined by Medvidovic and Taylor [MT00]: components, interfaces, connectors, and architectural configuration.

LRQ1.1: MontiArc component types fulfill the following requirements:

LRQ1.1.1: The MontiArc ADL provides several concepts to adapt subcomponents to the current context and to reuse component definitions.

- Subcomponents instantiate MontiArc components within a decomposed component. In this way, components can be reused within the context of another component without copy and paste. Declared subcomponents are validated by the context conditions R3, R4, R9, R10, R13, CV1, CV6, and CV7
- Generic components can be defined by declaring generic type parameters (see Listing 3.19). Generic types can be used as data types from ports and configuration parameters. Generics are consistently handled within all processing steps of MontiArc's tool chain. Context conditions R9 and R15 (see Section 3.5.3) support the valid definition and usage of generic components.
- Configurable components can be defined by declaring a list of configuration parameters (see Listing 3.19). Values can be assigned to these parameters when instantiating a configurable component. These parameters are consistently handled within all processing steps of MontiArc's tool chain. The context conditions R10 and R14 (see Section 3.5.3) help to use configurable components in a valid way.
- Component inheritance allows to define a component as an extension of a supercomponent (see Listing 3.19). A component inherits all model elements of the protected model interface (see Section 5.2.5) from its supercomponent. Consequently, the component interface as well as the architectural configuration of the supercomponent are inherited. Inheritance relations are validated by context conditions R11, R12, R14, and R15.

LRQ1.1.2: By using the MontiCore compilation unit concept, MontiArc components are organized in packages and have a qualified name (see Section 3.4.3). The qualified name is composed by combining the name of the package with the name of the component. This has the advantage, that the location of the compilation unit (file), which contains the component definition, and the component name are in a defined relation. A description of this concept is given in [GKR+06, Section 5.1].

LRQ1.1.3: The timing domain of a component can be selected within a component definition (see Section 3.3.1, Listing 3.29). The following timing domains are available. They are derived from the underlying FOCUS domains of timed, time-synchronous, and untimed streams [BS01, Chapter 4]:

- *Instant* components are time-aware and processes port-specific data events. Their results are emitted without delay.
- *Delayed* components are time-aware and process port-specific data events with processing time (delay ≥ 1).
- *Untimed* components are not aware of time and only process port-specific data events.
- Synchronous (*sync*) components are time-aware and synchronously process tuples of data events without delay and without explicit processing of time.
- Causal synchronous (*causalsync*) components are like synchronous components, but have delayed output.

LRQ1.1.4: Inner component definitions can be defined and automatically instantiated. This is only possible for inner components without generic type or configuration parameters. Since these concepts aim at component reuse, they are rarely used for inner component definitions. Automatic instantiation can be achieved by either using the optional instance

name of an inner component definition (see Listing 3.18) or by activating the `autoinstantiate` feature (see Section 3.3.3, Listing 3.28). The former is realized by transformation "Create Subcomponents from Named Inner Component Definitions", the latter by transformation "Autoinstantiate" (see Section 5.3.1).

LRQ1.1.5: The MontiArc ADL does not offer any means to share the state of a component directly with other components. Components solely interact via their component interface.

LRQ1.2: The component interface of a MontiArc component is described by a set of separate incoming and outgoing ports that are able to receive respectively send messages (see Section 3.2.2 and Listing 3.21). A port always has a type that determines the data type of the processable messages. According to requirement *LRQ1.4*, naming of ports is optional. Incoming and outgoing ports serve as connection point for incoming respectively outgoing FOCUS channels. The public model interface of a MontiArc component is defined by MontiArc's symbol table (see Section 5.2.5). It also contains information about generic type parameters and configuration parameters since both are needed to instantiate generic respectively configurable components.

LRQ1.3: The architectural configuration of decomposed MontiArc components is given by a set of `ArcElements` (see Listing 3.20) such as subcomponents and connectors. When instantiated as subcomponents, atomic and decomposed components are treated the same way. To ensure that the FOCUS properties of a channel holds, created connections are validated by the context conditions CO1-CO3, R1, R2, R5-R8, CV5, CV6, and CG1 (see Section 3.5).

LRQ1.4: Naming for ports and subcomponents is optional and their type and name can be used synonymously. The correct use of such implicit names is enforced by context conditions B1 and CV7 as well as MontiArc's symbol table.

LRQ1.5: The required Autoconnect feature is provided with two different strategies: `type` and `port` (see Section 3.3.2). It is realized by the transformation "Autoconnect Strategies" that is presented in Section 5.3.1. Thus, MontiArc provides means to automatically connect ports with the same unique name respectively type. This autoconnect mechanism can be controlled on the component level.

LRQ1.6: Constraints can be defined within components using OCL/P [Rum11, Chapter 3] and Java expressions. Both expression kinds have to evaluate to a Boolean expression. Since nonterminals of the respective language are embedded into the external nonterminal `InvariantContent` (see Section 3.4.4), MontiArc is adaptable to use further constraint languages. This is realized by using MontiCore's embedding mechanism (see [KRV08]).

LRQ1.7: MontiArc components, their interfaces, and contained elements can be documented with Java style comments. A documentation generator is provided that automatically derives JavaDoc like documentation from the contained comments (see Section 6.7). Thus, it is possible to document components and the contained model elements in an agile way.

LRQ1.8: A suitable mapping from MontiArc's basic architectural model elements to the Architecture Analysis and Design Language (AADL) is presented in Section 3.6. Language concepts, which are not present in the respective other language, are highlighted and possible emulations are discussed.

10.1.2 *LRQ2*: Usability

LRQ2.1: To increase the usability of MontiArc, a Maven plug-in is provided (see Section 5.7.2). It builds upon the MontiArc command line interface (CLI) and allows to process and validate MontiArc models, generates code, and automatically executes tests within a Maven build. To simplify the development of component libraries and to support distributed development with MontiArc, it further allows to built MontiArc projects on continuous integration (CI) servers.

LRQ2.2: MontiArc further provides an integrated development environment (IDE) that is realized as an Eclipse plug-in. It supports active specification (see [MT00]) of components with a context sensitive auto-completion. Depending on the context, available types (component as well as data types), connector sources and valid targets, and appropriate keywords are proposed. The documentation of library components (cf. requirement *LRQ5*) is integrated in the auto-completion to provide information about externally defined models.

LRQ2.3: Context conditions are defined in Section 3.5 which validate the correct usage of the provided architectural elements. Violations, which result in error or warning messages, are attached to the corresponding location in the editor of the IDE. In this way, direct feedback is provided to the modeler.

LRQ2.4: A quick start with MontiArc is eased by the contained wizard that sets up a completely configured MontiArc project. Thus, modeled components can be immediately analyzed, simulations can be generated, tests can be executed, and component documentation can be derived. Created projects are configured to automatically integrate predefined component libraries (see Section 6.8.2). Further wizards are provided to create new component types and component black-box tests (see Section 6.4.1).

LRQ2.5: Since the MontiArc IDE is realized as an Eclipse plug-in, all available distributed revision control clients for Eclipse can be used. MontiArc components are textual models. Thus, they can be managed by revision control systems, such as CVS[1], Subversion[2], and git[3], to support distributed development. Also, different versions of textual models can be easily merged and current changes can be automatically validated by using the Maven plugin-in. Further, agile and distributed development of components is supported. Changes in a component library, which influence depending projects, can be detected fast by using CI servers. This is easily possible, since the most CI servers, e.g., Jenkins[4], natively support the headless execution of the provided Maven plug-in (see Section 5.7.2). Headless execution means, that a build is executed without an available IDE and without user interaction.

LRQ2.6: To be used as an APL, an easy adoption and good usability is needed. To enable a quick start, Chapter 6 presents a complete tutorial with all required how-tos. By means of a running example, all aspects of the development of MontiArc components are introduced and explained. Beyond, further documentation is given about MontiArc's tools, libraries, the physical distribution of simulation components, and the integration of native implementations. Publicly available example projects[5] provide running examples for the tutorial and demonstrate

[1] The Concurrent Versions System, `http://www.nongnu.org/cvs/`
[2] Apache™Subversion®, `https://subversion.apache.org/`.
[3] Git –fast-version-control, `http://git-scm.com/`.
[4] Jenkins - An extensible open source continuous integration server, `https://jenkins-ci.org/`
[5] MontiArc Example Projects, `http://www.monticore.de/languages/montiarc/examples/`.

further aspects of the development of MontiArc components. Also, MontiArc has been used by students in several case studies. A subset of these are presented in Chapter 8.

10.1.3 *LRQ3*: Reusability and Extensibility

The MontiArc language and the corresponding tooling is designed to be extendable, and thus adjustable to different domains. This is achieved by fulfilling the following requirements.

LRQ3.1: A structured method that addresses language and tool extensions is presented in Chapter 7. Several languages that adjusted MontiArc to different domains are discussed in Chapter 9. It is shown that architectural elements can be added and refined to allow new analysis or adjust the language.

LRQ3.2: MontiArc does not provide means to describe respectively define the behavior of atomic components itself. The concrete behavior is either defined by decomposition or external behavior descriptions (see requirement *SRQ2*). Nevertheless, it is possible to embed behavioral descriptions directly into atomic component definitions, e.g., to adjust MontiArc to a specific target language. For this, a predefined extension point is provided which allows to embedd behavioral descriptions defined in another behavior description language.

LRQ3.3: Modularity of the MontiArc ADL and the corresponding tools is achieved by a strict separation of language frontends, generators, and the RTE into separated Java projects. Due to MontiCore's context condition framework and MontiArc's transformation framework (see Section 5.3.3), even single context conditions or transformations can be reused in extended languages.

10.1.4 *LRQ4*: Type System

LRQ4.1: MontiArc's symbol table provides a concrete type system for architectural elements. It also contains an abstract type system for object-oriented data type definitions (see Section 5.2). The developed context conditions validate connections according to this abstract type system. The transformations and generators only operate with these abstract types. To integrate a concrete data type system, suitable adapters have to be integrated (see Section 7.3.2).

LRQ4.2: As a default realization of the abstract type system, MontiArc reuses the type system of class diagrams (CDs) defined by the UML/P [Rum11, Rum12, Sch12]. Thus, data types used within an architectural model can be defined in CDs. Since Java is the target language of the simulation generators, a second default realization is given by reusing Java's type system. Hence, it is also possible to define classes, interfaces, and enumerations in Java and use them as data types within MontiArc components. Both data type systems are adapted to the abstract data type system using language aggregation techniques described in [Völ11, HLMSN+15]. Please note, both type systems can be used in parallel. In this way, data types referenced within a single model can be partially defined in Java classes and in CDs.

10.1.5 *LRQ5*: Libraries

MontiArc components can be packaged into component libraries which can be reused in distinct modeling projects (see Section 6.8.1). However, reuse on an architectural layer imposes certain

requirements on the underlying ADL which are defined by McVeigh et al. [MKM06] (Alter, NoImpact, NoSource, and Upgrade). The latter two are consolidated into the following subrequirements. The former are fulfilled by the MontiArc ADL. MontiArc library components can be either extended using inheritance or adapted in a dedicated decomposed component. Alternatively, if the sources of a library are available, components can be copied, altered, and released as a new (local) version of the library. In this way, library components can be altered to adapt them to a new context. These alterations do not require changes of the original model and, thus, do not influence other users of a library (NoImpact).

LRQ5.1: MontiArc libraries are released as a set of Maven artifacts which have a unique identifier and a version number (project coordinates). The unique identifier is given by the library's group ID in combination with its artifact ID. An existing library can be simply integrated into a MontiArc project by adding its coordinates to the dependencies of the project (see Section 6.8.4). Alterations of a MontiArc libraries are released with a new version number. Thus, updates of a library can but must not be accepted. If an update is accepted and an altered component has been changed, the aforementioned alteration technique, which reuses inheritance or adaptation, allows to adjusted the altered component to the incoming changes.

LRQ5.2: MontiArc's tooling is based on the underlying MontiCore DSLTool framework. Since MontiCore also provides model loading from jar files, reuse of library components is transparent for the user.

LRQ5.3 The structure of MontiArc libraries strictly separates source and binary artifacts which are contained in distinct released library artifacts. To reuse a library, at least its binary artifacts have to be available. In this way, the intellectual property of closed source libraries can be retained. Since the integration of libraries is realized on a technical layer, the modeler can reuse components from a closed source library in exactly the same way as components from a library with available sources.

10.2 Simulation Requirements

The developed FOCUS simulation allows to explore and validate MontiArc component models in an agile way. In combination with the MontiArc ADL, it can be used as an APL which is suitable to rapidly prototype system models. According to Baumeister et al. [BHH+06], an APL integrates architectural elements into a programming language. Since MontiArc is mainly an ADL and not a programming language, this is achieved in another way that still offers the advantages of architectural programming, i.e., prevention of architectural erosion and enforcement of the architecture on the programming level.

The former is achieved since MontiArc models are the primary development artifacts when programming with MontiArc. The provided generator (see Section 5.4.2) produces abstract implementations for atomic components which have to be extended to implement the concrete behavior in Java. Consequently, evolution has to be primary performed on the architectural layer which immediately affects the programming layer. This is possible since the generated code is not manually adjusted (see Section 5.5.2) and an agile build infrastructure is provided (see Section 5.7.2). Further, the architecture is enforced on the programming layer using a generative approach (see Section 5.4). Architectural elements are available in the implementation by pro-

viding a well defined application programming interface (API) with architectural elements. In this way, implementing the behavior of an atomic MontiArc component corresponds to architectural programming.

10.2.1 *SRQ1*: Platform Independence

The MontiArc simulations is executable on different platforms without the need for specialized compilation. The selected implementation language Java is a platform independent, object-oriented general purpose language (GPL) that translates sources into platform independent bytecode. The bytecode is then interpreted by platform specific implementations of the Java Virtual Machine (JVM). Thus, it has to be clarified whether the developed simulation RTE is compatible with many JVMs.

The most common JVMs, such as Oracle's Java SE[6] or OpenJDK[7], support a rich set of APIs and concepts. However, several JVMs are tailored to embedded and mobile devises such as the Java Platform Micro Edition (Java ME)[8] or JamVM[9]. Since the embedded target hardware has less resources than regular computers, these JVMs typically do not support the full range of APIs. A standard for embedded JVMs is the Connected Limited Device Configuration (CLDC) [JSR14]. It defines precisely which classes of Java's default API have to be supported by embedded JVMs. While previous versions of the CLDC have not supported the Collections library, the most recent version does. In contrast, the Reflection APIs is still not supported.

The developed simulation framework uses Java Collections and avoids the Reflection API. Since the RTE and generated classes do not depend on external dependencies, compatibility to the most recent CLDC is ensured. The developed RTE has been tested on Java SE 6 – 8 and open JDK 6 and 7 on windows and linux computers. It has also been tested on a JamVM in combination with the GNU classpath[10] executed on a Raspberry Pi model B[11]. Consequently, requirement *SRQ1* is fulfilled.

10.2.2 *SRQ2*: External Component Implementation

The behavior of atomic MontiArc components is implemented externally and the implementation can be flexibly exchanged before a simulation is executed. Atomic MontiArc component models do not contain a reference to the concrete implementation. Consequently, there is no hard-wired dependency between the component model and its implementation.

The developed RTE follows a strict separation of interfaces and implementations. All classes of the RTE (see Section 4.2) and all classes produced by MontiArc's generator (see Section 5.4) only have dependencies to interfaces. Concrete instances are created by factories (see Section 5.5.2) that are also used to integrate the implementations of atomic components into the

[6] Java SE at a Glance, `http://www.oracle.com/technetwork/java/javase/overview/index.html`.
[7] OpenJDK, `http://openjdk.java.net/`.
[8] Java ME, `http://www.oracle.com/technetwork/java/embedded/javame/index.html`.
[9] JamVM, `http://jamvm.sourceforge.net/`.
[10] GNU Classpath, `https://www.gnu.org/software/classpath/`.
[11] Raspberry Pi, `http://www.raspberrypi.org`.

simulation.

To interchange concrete implementations of interfaces, direct instantiation of objects using the `new` statement is completely avoided. Thus, generated classes for decomposed components use contained subcomponents only via their generated Java interface. Object instantiation of subcomponents is handled using generated singleton factories [GHJV95]. By creating custom component factories that instantiate alternative implementations of atomic components, component implementations can be exchanged (see Section 5.5.2). The same method allows to exchange implementations of decomposed components with stubs (see Section 6.4.2).

10.2.3 *SRQ3*: Mathematical Foundation

The MontiArc simulation implements the communication model of FOCUS. Outgoing ports directly transmit data and tick messages to connected incoming ports. A record of these transmissions directly corresponds to a FOCUS stream that describes the timed communication between sender and receiver. Since streams are mostly not needed during the execution of a simulation, streams are not recorded by default to reduce the amount of allocated memory (see requirement *SRQ10*). For testing and analyses, several ports can be flexibly replaced with test ports which explicitly record transmitted messages in a stream data structure that can be analyzed during or after the execution of a simulation. Examples that demonstrate how to test MontiArc components are given in Section 6.4.

Even though the simulation is executed in a single thread with synchronous blocking method calls, atomic components are implemented in an event-based manner. Therefore, component implementations have to provide suitable methods which are automatically called by the scheduler to stimulate components with incoming events as described in Section 4.4. Produced messages are then transmitted using the `send()` methods from the outgoing ports of the component's interface. Although the following message transmission and possible event propagation by the scheduler takes some real time, no simulation time has past when the control flow returns to the component. Consequently, the simulation is logically asynchronous and event-based.

In conformity with FOCUS channels, connections in the simulation are unidirectional and connect one sender with one or more receivers. This structural property is granted by the developed RTE (see Section 4.2.2).

A FOCUS component encapsulates its internal state and exclusively interacts with connected components by exchanging messages. On a structural layer, this is enforced by the provided RTE and the generated component classes. However, since atomic components are implemented manually, it is possible to implement dirty components (see Definition 5.1). These share the state of all component instances by introducing static fields which can be even accessed by other component implementations. Less obvious violations of this property can be achieved if side effects are produced outside the state space of the component, e.g., by manipulating the file system. This problem is encountered in two ways. First, dirty components must be marked with the `@sideEffects` annotation together with a detailed description of the side effects (see Section 6.7). If dirty components are not explicitly marked, the user of such a component is not able to predict the side effects. This is especially harmful since dirty communication bypasses the simulation time. Second, guidelines for the development of dirty components are given in Section 5.5.3.

Beyond, analyses of the handwritten source code could be executed to identify malicious communication. Therefore, additional input is needed which classifies classes or packages of the Java API as safe or unsafe. First experiences with the Architecture Alignment Checker (see Section 9.1) lead to promising results. However, an automatic analysis of atomic component implementations is not yet integrated into the MontiArc tool chain.

10.2.4 *SRQ4*: Component Timing Classification

The MontiArc simulation supports components of all timing domains defined in Section 4.4. Components process an event trace which is produced from the incoming channels based on their timing domain. The underlying simulation and scheduling is based on timed streams that correspond to channels in FOCUS. The concrete mapping from message streams to event traces is realized by the generated code as described in Section 5.4.2. Time and data events (see Definition 4.4 and 4.3) are propagated to components with instant or delayed timing. Data event tuples (see Definition 4.5) are propagated to synchronous and causal synchronous components. Untimed components solely process data events.

10.2.5 *SRQ5*: Simulation Time

The simulation time is realized by using FOCUS streams to simulate interaction between components (see Section 4.1). In FOCUS, time is modeled with logical messages called ticks ($\surd$). The developed simulator simulates the flow of ticks through the simulated system with explicit tick messages. Ticks are synchronized by the scheduler (see Section 4.3.2) to increase the local clocks. Consequently, the simulation time represented by ticks is decoupled from the real time and requirement *SRQ5* is fulfilled.

10.2.6 *SRQ6*: Distribution

A common approach to simulate event-bases system is the implementation of a global event queue [DIHK$^+$01, Chapter 15]. Events are annotated with receiver time stamps, the producer, and the receiver of the event. If an event is produced, it is added to the global event queue that is sorted according to the receiver time stamps. A global clock is increased to the next recent receiver time stamp and the corresponding event is propagated to the receiver. Synchronizing a global event queue of a distributed simulation leads to additional overhead which is introduced by managing the events of all distributed nodes within a single queue. Consequently, the node which hosts the global event queue is a bottleneck that routes the communication between all nodes [Cra96].

In contrast, the event-based MontiArc simulation directly transmits all data and tick messages from a sender to the connected receiver. The synchronization of the simulation time is performed locally at the incoming ports of a component (see Section 4.4, Section 4.3). Consequently, no further synchronization overhead is needed between distinct nodes of a distributed simulation and requirement *SRQ6* is fulfilled. A process model to derive distributed simulations and a small case study is given in Section 6.9.

10.2.7 *SRQ7*: Component Testing

The behavior of MontiArc components is testable using the developed simulation. This comprises a deterministic execution of the simulator as well as testable black-box, white-box, and timing behavior of components.

SRQ7.1: A deterministic execution of a simulation is guaranteed if the scheduling, the message transport, and all simulated components are deterministic. A deterministic component always generates the same output for equal received input streams. Two streams s_1, $s_2 \in M^\omega$ are equal iff both are a prefix of each other ($s_1 \sqsubseteq s_2$ and $s_2 \sqsubseteq s_1$, see Table 4.1). Thus, equal streams contain the same messages and ticks in the same order. The mappings defined in Section 4.4 conceptually allow multiple valid data event traces for two or more given input streams. This conforms to the FOCUS property, that the order of messages contained in the same time interval of two different streams is not determined [BS01]. Consequently, a deterministic component has to produce the same results for the same input streams, even if data events received from distinct ports within the same time interval are propagated in a different order. However, a deterministic scheduler always propagates the same events in an identical order for an identical simulation input.

In contrast to the modeled target system, components in the MontiArc simulation run in a single thread and the execution order of the components is determined by the scheduler. Assuming that only deterministic components are involved in a simulation, also the scheduler has to be deterministic to ensure deterministic results. This can be arguable affirmed with the following considerations:

- The simulation and all contained components always have the same state after the setup. This is guaranteed since the setup code does not change and the structure of the simulated system is static.
- Since the input streams of the simulation are always the same, only the order of propagated data events within the distinct time frames may differ during multiple executions.
- Because only deterministic components are involved, the order of input events does not matter. Consequently, the involved components always produce the same output.
- If a data message is received on a port that is blocked by a tick, the data is buffered until the port is reactivated.
- Ticks are used to synchronize the distinct input channels of a component. Since the input streams of the repeated simulation are always the same, tick scheduling always leads to the same time events in the event trace.
- After propagating a time event to the component, the blocked ports are reactivated to schedule buffered messages. The reactivation order might influence determinism. Since the scheduler stores the ports of a component in a list (see attribute `comp2Port` in Figure 4.12), they always have the order in which they are inserted into the list. This is done during the setup of the components which does not change between two executions of the simulation. Hence, the activation of ports is performed in a deterministic order that does not change during distinct executions of the simulation.

SRQ7.2: Since the simulation scheduler is deterministic, the behavior of components is testable. JUnit [www13c] is well suited to implement automatically executable component tests.

For this, the component under test is set up with a scheduler and the outgoing ports are allocated with test ports. Then, the incoming ports are fed with input streams that stimulate the component. The produced output messages are then stored in the streams of the outgoing test ports. These are compared with the expected results. To simplify the creation of black-box tests, MontiArc provides a stream based test language that automatically generates JUnit tests from input and expected streams. The test language is presented in Section 6.4.1.

SRQ7.3: White-box tests aim at validating interaction patterns between subcomponents of a decomposed component. Subcomponents of a decomposed component can be instrumented with test ports, too. In this way, the information flow within a decomposed component can be analyzed. Methods and techniques to implement white-box tests for MontiArc components are explained in Section 6.4.2.

SRQ7.4: Generated decomposed components only have a loose dependency to the Java interfaces of their contained subcomponents. Thus, subcomponents, which are irrelavant for the current test or have a non-deterministic behavior, can be exchanged with mock components.

SRQ7.5: Since the developed stream data structure supports all operations given in Table 4.1, the timed behavior of a component under test can be validated.

Please note that it is also possible to realize specialized scheduler strategies for testing since the simulation scheduler could be exchanged (see requirement *SRQ9.2*). For example, determinism of components (see above) can be tested by using a scheduler that deterministically synchronizes ticks, does not change the order of events on a single port, but varies the event propagation order between distinct ports of a component. Using such a scheduler in repeatedly executed simulations, deterministic components still have to generate the same output for the same simulation input. If the output varies between different executions, the tested component is not deterministic.

10.2.8 *SRQ8*: Extensibility

A consequent use of Java interfaces within the RTE and the generated code decouples concrete implementation classes. Using object oriented inheritance and substitution, existing implementations can be easily replaced with type compatible extensions (see Section 7.2). The extendable generator infrastructure allows to integrate the created extensions into the generated simulation classes (see Section 7.2.4).

10.2.9 *SRQ9*: Scheduler

SRQ9.1: MontiArc provides a default scheduler implementation which can be used to execute a logically asynchronous, event-based simulation of MontiArc components. The implemented scheduling strategy is presented in Section 4.3.

SRQ9.2: As already mentioned, Java interfaces are consequently used to define the interfaces of generated and RTE classes. Hence, it is possible to flexibly interchange the concretely used implementation of an interface. Therefore, a custom scheduler can be realized which implements the scheduler interface `IScheduler` and an instance can be passed to the `setup` method of the component that is to be simulated (see Figure 4.10 on page 94). Actually, this technique has been used to compare distinct scheduler implementations in Section 4.5.

SRQ9.3: MontiArc restricts the number of schedulers for a single component to one. Nevertheless, each component instance can be controlled by a dedicated scheduler instance. Therefore, override the `setup()` method within an atomic component implementation, ignore the passed scheduler object, instantiate the dedicated scheduler, and call `super.setup()` with the dedicated scheduler instance. This technique can also be used to replace the scheduler of all subcomponents of a decomposed component. In this case, also a dedicated factory has to be implemented which instantiates the adjusted decomposed component instead of the original version.

10.2.10 *SRQ10*: Optimizations

Runtime and memory inefficiencies have to be avoided. Several design decisions are influenced by this requirement. Further, several optimizations of the scheduler or the generated code have been performed. These are:

- No dedicated representation of a connector exists in the RTE. Since the semantics of a FOCUS channel (immediate, reliable, and order preserving) corresponds to a direct connection of outgoing and incoming ports, connector objects are superfluous.
- Port instances are only created for incoming ports. Since the default port implementation can act as incoming and outgoing ports (see Figure 4.10), the incoming ports of a receiver are used as outgoing ports from the sender. This is automatically realized during the setup of decomposed components.
- The internal data structures of the default scheduler have been replaced after a detailed comparison of alternatives in Section 4.5.
- Further redundant objects have been reduced for certain component patterns (see Section 5.6). This especially comprises the reduction of redundant forwarding ports and the reuse of a central tick object.

10.3 Conclusion

The thesis at hand poses the research question: How to design an ADL that copes with the impediments of architectural modeling in practice? The approach to answer this questions was, first, to analyze existing ADLs which target at the domain of interactive systems and identify, whether an existing modeling language can be extended to fulfill the derived requirements. Also APLs have been examined since they are suitable to rapidly prototype architectures in early development stages. Since no language fulfills all defined requirements, MontiArc has been developed as a case study to explore and evaluate concepts to answer the posed question. By summarizing own experiences and impediments documented in the literature, the following subquestions have been derived.

***RQ1* How to design a lightweight and easy to learn ADL?** This question is addressed by defining an ADL which contains only a restricted set of mandatory architectural elements. According to Medvidovic and Taylor [MT00], these elements are components, ports, connectors, and the possibility to define architectural configurations. To be easy to learn, a language has to be ready to use. In contrast to other extendable ADLs with a reduced set of modeling elements, such

as Acme [GMW97] or xADL [DvdHT01], MontiArc has a concrete architectural style based on a formal mathematical model named FOCUS [BDD+93, BS01]. This formal foundation gives MontiArc models a strong semantics and MontiArc is ready to use without the need for extension.

Furthermore, an IDE is necessary to assist new users with the syntax and semantics of the language. Features known from GPL IDEs, such as keyword highlighting, automatically executed syntax checks and context conditions, or active modeling, are suitable to achieve a fast rising learning curve. Most of these features, however, are more suitable for textual than graphical modeling languages.

***RQ2* How to design an extendable ADL which allows to reuse as much tooling as possible?** As discussed in this thesis, several extendable ADLs exist in practice. Among them are languages such as the Unified Modeling Language (UML) [OMG11b], the System Modeling Language (SysML) [OMG12], the AADL [FGH06], Acme, or xADL. While the syntax of all these languages can be extended, a following extension of the corresponding tooling results in high effort or incompatibility to the original language.

The research question is answered in this thesis by the definition of a structured method for ADL extensions. It covers extensions of the tooling such as analyses, transformations, or simulations, and is based on the language extension and combination techniques discussed in [Völ11, HLMSN+15]: language aggregation, language embedding, and language inheritance.

However, several preconditions have to be met to use all proposed extension methods. First, the underlying language development framework has to support the named extension techniques. If, e.g., language inheritance is not supported, resulting copy and paste extensions are hard to maintain and new features or bugfixes of the base language cannot be easily transfered to the extension. Also the underlying technical infrastructure has to be designed in an extendable way. Thus, it has to be possible to flexibly exchange language processing components. In MontiArc, this is achieved with dependency injection and the consistent use of implementation interfaces. Further, it prove successful to design context condition checks and transformations in a way that allows to modularly add or remove single checks or transformations. In doing so, checks for new model elements can be added easily while reusing the predefined context checks from the superlanguage.

Another important aspect of extensibility is the needed effort to create an extension and the modularity of the language infrastructure. Some of the analyzed languages require to checkout the complete language infrastructure and compile it together with the extension. In contrast, MontiArc can be extended by simply defining the needed dependencies in the build system. Finally, extension points of the language as well as the tooling have to be documented. If a documentation is missing, developers have to identify these extension points in the source code themself, which is a very sophisticated task.

***RQ3* Which concepts can be applied to an ADL to support reuse of architectural models?** This question is addressed by three aspects. First, the language itself has to provide concepts that alleviate reuse. In MontiArc, this are:

- Separation between component type definition and component instantiation. It allows to reuse defined components as subcomponent in an object oriented way.
- Means to configure component definitions when they are instantiated. This comprises

generic and configurable components.
- Component inheritance allows to refine existing component definitions with a richer interface and functionality.
- Also delta modeling techniques [CHS10] adopted by Δ-MontiArc [HRRS11, HKR+11] are well suited to reuse component models in component variants.

Second, the underlying technical infrastructure has to support reuse in a controlled way. A well defined library concept is suitable to ease reuse. Therefore, the library requirements defined by McVeigh et al. [MKM06] - Alter, NoImpact, NoSource, and Upgrade (see requirement *LRQ5*) - have to be fulfilled by an ADL. By using the modelpath concept of MontiCore and a strict separation of released library artifacts (see Section 6.8.1), MontiArc fulfills these requirements.

Third, the documentation of components is an important factor to ease reuse. The interface, the state space, and the behavior of a component have to be documented. Further, the documentation has to be seamlessly integrated into the IDE and a provided library. In this way, black-box library components can be reused, even if the source models are not available.

***RQ4* How to integrate agile development methods with architectural modeling to allow for incremental modeling and early validation of the architecture?** Again, this question is addressed by multiple aspects of the thesis respectively MontiArc. The early validation of architectures is supported by automatically executed model analysis and model simulations. The former guarantees well formed models and that connections are correct according to the type system. The latter allows to explore and validate the behavior of architectural prototypes.

To speed up the development effort of architectural prototypes, the following agile development concepts have to be provided by an agile ADL:

- Testing: Automatically executable tests used as regression tests. This way, negative impact of changed models and behavior implementations to other parts of the architecture can be detected. This comprises component black-box and white-box tests.
- Automatic builds: By providing a headless build infrastructure which can be executed outside the provided IDE, component tests can be automatically executed on continuous integration servers such as Jenkins. This is especially useful in a multi-project setup. If a single project is changed, not all projects need to be tested on the local machine.
- Distributed development: Models need to be manageable by a revision control system (RCS). Graphical models or textual XML based models are less suited for standard RCSs since model merging is not easily possible and additional effort has to be taken. Since pure textual models can be simply merged with a text-based diff, textual modeling languages are better suited for distributed development. Additionally, automatic builds and testing needs to be available to automatically validated merged models afterwards.
- In place documentation: Agile methods avoid a documentation based development process. Needed documentation is given as source code comments directly in the code. If the code is changed, the documentation can be aligned as well. MontiArc adapts this concept and uses comments and documentation tags to automatically generate component documentations. As already mentioned, these documents are also integrated in libraries and the MontiArc IDE.

By elaborating the posed research questions, this thesis contributes to two areas of research:

language engineering and architectural modeling. It provides a consistent case study for:

1. Language aggregation and extension with MontiCore's symbol table framework [Völ11, HLMSN+15].
2. Development of modular context conditions for model analyses [Völ11, Sch12].
3. Development of extendable code generators based on [Sch12].
4. Development of model libraries with the MontiCore framework.
5. Development of a modeling IDE based on [KRV07a].

Thus, it contributes to the domain specific language (DSL) toolbench MontiCore [GKR+06, Kra10] by evaluating developed concepts. Experiences made during the development of MontiArc have been reflected back into the named concepts.

Furthermore, it contributes to the research area of architectural modeling by providing an ADL that copes with the impediments of architectural modeling in practice (see [WH05, Woo05, Pan10, MLM+13]. Language engineering and agile methods have been applied to an ADL. In this way, an easy to learn ADL with an IDE and headless build tools have been designed. It can be integrated into existing tool chains; designed models can be incrementally developed. It comes with a simple, but mathematically founded, architectural style which can be easily adjusted and extended. Also, the component library concepts defined by McVeigh et al. [MKM06] have been implemented, evaluated, and elaborated. Finally, the methods of architectural programming, as for example given in ArchJava [ACN02b] or JAVA/A [BHH+06], have been adopted and adjusted.

The concepts developed for MontiArc can be adopted by ADLs from the community. This especially comprises the following aspects:

- Data type system: In practice, a huge amount of data types is used in distributed applications. For example, in the automotive domain, a distinct data type is most often defined for each message channel. Thus, there is quite some effort to define data types in a proprietary type definition language of an ADL and in the target GPL. Also, both type definitions have to be kept aligned with each other. Thus, an ADL should have the ability to process externally defined data types. Until now, none of the observed related languages (see Section 2.3) provides this feature.
- Automatic and continuous analyses: To support incremental and collaborative modeling, automatically executable analyses are suited to continuously analyse a model and detect inconsistencies. Especially headless executions of the ADL tools without user interaction are well suited to automatically analyse models on a CI server. In this way, a model can be incrementally extended by different developers. Some observed languages already provide tools with a CLI which can be used for this purpose. Others depend on user interaction to, e.g., execute analyses and tests.
- Extendability: Two aspects are important for an extendable ADL. First, feasible extensions have to be identified and documented. Especially, the dependencies between different extensions have to be clarified. If, e.g, a new language element is to be added, model analyses, transformations, and code generators have to be extended as well. Especially, such extensions have to be encapsulated in a separated module without the need for adjusting the language core or depending on its source code.
  Second, the effort to create an extension has to be low. Off course this also comprises

a good documentation of the extension points and extension mechanisms. Nevertheless, also the reuse and combination of existing languages with an ADL in a component based way yields to a saving of time. If well tested languages are used for such a combination, the quality of the result can also be increased compared to a development from the scratch. Therefore, however, the underlying language workbench has to support the extension methods: language aggregation, language embedding, and language inheritance.

- Timed FOCUS simulation: MontiArc contains scheduling algorithms for the timed simulation of FOCUS models. Also the combination of different timing domains is handled by the MontiArc simulation. These techniques can be especially interesting for AutoFocus 3 [HST10, HF10, www14f] since it shares the mathematical foundation with MontiArc but is restricted to the simulation of time-synchronous streams.
- Model libraries: Almost all observed ADLs allow to define or reuse models in libraries. However, none of these languages fulfills the component library requirements defined by McVeigh et al. [MKM06]. Especially the ability to handle closed-source libraries is missing in all regarded library concepts. Also a concrete versioning and dependency management is missing. As demonstrated by MontiArc, using software build tools for this purpose, such as Maven, is expedient. Nevertheless, a well defined structure of the released library artifacts is mandatory.

Regarding the current state of MontiArc and the experiences made during the development, additional research might be done in the area of language engineering.

- Generator interfaces: The MontiArc code generator is developed in an extendable way. It provides extension points which can be used to integrate new code generator templates or replace existing code generator templates with new ones. Nevertheless, the underlying framework does not validate whether added generator templates are suitable for the used extension point. A generator interface, as for example mentioned by Roth et al. [RR15], can be used to compose generators in a component based way. It has to define a) which source model elements are passed to an extension point and b) which target code respectively model elements have to be produced by a generator which is added to an extension point. Also the configuration of generators can be defined in a product-line way using feature diagrams. Together, composed generators can be analysed for validity.
- Symbol table generation: The used symbol table infrastructure provided by MontiCore based on the concepts presented in [Völ11, HLMSN+15] are well suited to realize language reuse and composition. Nevertheless, the effort to implement the needed implementation artifacts can be reduced drastically by generating parts of the symbol table. While parts of the symbol table can be easily generated from the language defining grammar, language composition requires new concepts for the automatic synthesis of the required implementation artifacts.
- IDE extensions: MontiArc allows to reuse large parts of the implemented tooling for languages which extend MontiArc: context conditions, symbol table components, transformations, the underlying transformation framework, and code generators. However, it is not possible to directly extend the corresponding MontiArc Eclipse editor. To reduce the effort of developing IDEs for sublanguages, research in this direction is needed.

Also some further research in the area of architectural modeling and ADLs might be done:

- Native casestudy: A suitable pattern and a proof of concept for the simulation of native C components is presented in Section 6.3.2. This technique could be validated in a larger case study, e.g., in the embedded domain. Evaluating the simulation performance in such a setup, the reuse and integration of existing libraries, and the integration of object oriented native languages remains to be done. The latter is especially interesting since it solves the issue with the automatically shared state of C components.
- Architectural alignment: The Architecture Alignment Checker (AAC) mentioned in Section 9.1 allows to automatically check the consistency between a Java implementation of a system and its architectural description specified using MontiArc. The presented concept might be extended in multiple ways. First, it could be transfered and adapted to the domain of information flow architectures which is more suitable for architectural descriptions in MontiArc. Second, an agile development method might be developed which allows to continuously validate the architectural conformance of systems and keeps the effort of aligning architecture and code low. Therefore, a seamless integration into build tools which are used on continuous integration servers is mandatory. Third, also an integration into IDEs is promising. Automatically marking forbidden dependencies and communications based on the architectural model immediately alerts developers that their implementation is not conform with the architecture. Also, the autocompletion functionality can be extended to only propose elements, such as classes, fields, or methods, which are conform to the architecture in the current context. Finally, the AAC consistency checks are based on a manually created mapping from architectural elements to elements of the target language. Techniques to (semi-) automatically derive such mappings from a given architecture and implementation will reduce the effort to apply architecture alignment checks to a running project.
- Aggregated timing: In the MontiArc simulation, all ticks mark the end of a logically equidistant time interval. If distinct time resolutions within a system are of interest, always the highest time resolution has to be chosen for the complete system simulation. Thus, if parts of a system produce events every millisecond, and other parts produce events every hour, a logical time interval corresponds to a millisecond. In the simulation, this leads to many unnecessary time events for components with a coarser time resolution. To speed up the simulation, consecutive time intervals which do not contain any message could be aggregated and scheduled together. To schedule simulations with distinct time resolutions, new scheduling strategies have to be developed.

Appendix A

Index of Abbreviations

AADL	Architecture Analysis and Design Language
ABP	Alternating Bit Protocol
ACC	adaptive cruise control
AD	activity diagram
ADL	architecture description language
API	application programming interface
APL	architectural programming language
ArcD	Architecture Diagram
ARP	Address Resolution Protocol
AST	abstract syntax tree
CD	class diagram
CI	continuous integration
CLI	command line interface
DET	data event tuple
DNS	Domain Name System
DSL	domain specific language
EBNF	Extended Backus-Naur Form
GPL	general purpose language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	integrated development environment
IP	Internet Protocol
JNI	Java Native Interface
JVM	Java Virtual Machine
LAN	local area network
LHS	left-hand side
LRQ	language requirement
MAC	media access control
MDA	Model-Driven Architecture
MDD	Model-Driven Development
OCL	Object Constraint Language
OD	object diagram
OMG	Object Management Group
RARP	Reverse Address Resolution Protocol

RCS	revision control system
RHS	right-hand side
RRS	round robin scheduler
RTE	runtime environment
SC	statechart
SD	sequence diagram
SPL	software product line
SUD	system under development
SysML	System Modeling Language
TCP	Transmission Control Protocol
UML	Unified Modeling Language

Appendix B

Diagram and Listing Tags

Stereotype	Description
≪abstract≫	An abstract class in a CD.
≪artifact≫	A produced or consumed file or object in an AD.
≪creates≫	Denotes that one element creates another.
≪delayed≫	The annotated connector delays transmitted messages.
≪enum≫	An enumeration in a CD.
≪gen≫	Generated element.
≪handcoded≫	Handcoded element.
≪interface≫	An interface in a CD.
≪uses≫	Denotes that one element uses another.
≪workflow≫	Denoted activity in an AD is implemented as a workflow.

Table B.1: Explanation of the used stereotypes within listings and tags.

Tag	Description
AD	Activity Diagram
AJava	AJava Diagram
CD	Class Diagram
I/O-Test	I/O Test Definition
Java	Java Source Code
MA	MontiArc Diagram
MA^{PR}	MontiArc with Property Extension
MA^{SC}	MontiArc with SC Extension
MAA	MontiArcAutomaton Diagram
Meta-CD	Metamodel Class Diagram
MG	MontiCore Grammar
MLD	MontiCore Language Definition
Nsp.-OD	Namespace Object Diagram
pom.xml	Maven Project Configuration
Product-CD	Class Diagram of the Resulting Simulation Product
RT-OD	Runtime Object Diagram
RTE-CD	RTE Class Diagram
Symtab-OD	Symbol Table Object Diagram
VC	Value Chain Diagram (ProNet^{sim})
VCCfg	Value Chain Configuration (ProNet^{sim})
...	Annotates an incomplete diagram.

Table B.2: Explanation of the used tags in listings and figures.

Appendix C

Grammars

C.1 Architectural Diagrams Grammar

MG

```
1  package mc.umlp.arcd;
2
3  version "$LastChangedDate: 2014-06-05 15:31:04 +0200 (Do, 05 Jun
4      2014) $, $LastChangedRevision: 2862 $";
5  /**
6   * Grammar for common architectural elements. Provides
7   * infrastructure for component definitions, component
8   * interface definitions, and the hierarchical structure
9   * of components.
10  *
11  * @author Arne Haber
12  *         (last commit) $LastChangedBy: ahaber $
13  * @version $LastChangedDate: 2014-06-05 15:31:04 +0200 (Do, 05
14  * Jun 2014) $
15  *          $LastChangedRevision: 2862 $
16  */
17 grammar ArchitectureDiagram extends
18     de.monticore.lang.common.CommonValues {
19
20 /* ======================================================*/
21 /* ====================== OPTIONS =======================*/
22 /* ======================================================*/
23   options {
24     compilationunit ArcComponent
25     nostring
26     parser lookahead=5
27     lexer lookahead=7
28   }
29
30 /* ======================================================*/
31 /* ==================== PRODUCTIONS =====================*/
32 /* ======================================================*/
33   /**
34   * A component may contain arbitrary many ArcElements.
35   * This interface may be used as an extension point to
```

```
36  * enrich components with further elements.
37  */
38  interface ArcElement;
39
40  /**
41   * External to embed languages that allow implementing
42   * component behavior.
43   */
44  external BehaviorEmbedding;
45
46  /**
47  * A component is a unit of computation or a data store.
48  * The size of a component may scale from a single
49  * procedure to a whole application. A component may be
50  * either decomposed to subcomponents or is atomic.
51  *
52  * {@attribute stereotype an optional stereotype}
53  * {@attribute name type name of this component}
54  * {@attribute instanceName if this optional name is given,
55  *   a subcomponent is automatically created that
56  *   instantiates this inner component type. This is only
57  *   allowed for inner component definitions.}
58  * {@attribute head is used to set generic types, a
59  *   configuration and a parent component}
60  * {@attribute body contains the architectural elements
61  *   inherited by this component}
62  */
63  /ArcComponent implements
64      (Stereotype? "component" Name Name? ArcComponentHead "{")=>
65          ArcElement =
66    Stereotype?
67    "component" Name (instanceName:Name)?
68    head:ArcComponentHead
69    body:ArcComponentBody;
70
71  /**
72  * A components head is used to define generic type
73  * parameters that may be used as port types in the
74  * component, to define configuration parameters that may
75  * be used to configure the component, and to set the
76  * parent component of this component.
77  *
78  * {@attribute genericTypeParameters a list of type
79  *   parameters that may be used as port types in the
80  *   component}
81  * {@attribute parameters a list of ArcParameters that
82  *   define a configurable component. If a configurable
83  *   component is referenced, these parameters have to be
84  *   set.}
```

```
85  * {@attribute superComponent the type of the super
86  *   component}
87  */
88  /ArcComponentHead =
89    (options {greedy=true;}:
90      genericTypeParameters:TypeParameters)?
91    ("[" parameters:ArcParameter
92    ("," parameters:ArcParameter)* "]")?
93    ("extends" superComponent:ReferenceType)?;
94
95  /**
96  * The body contains architectural elements of
97  * this component.
98  *
99  * {@attribute arcElement list of architectural elements}
100 */
101 /ArcComponentBody =
102   "{"
103     ArcElement*
104   "}";
105
106 /**
107 * An ArcInterface defines an interface of a component
108 * containing in- and outgoing ports.
109 *
110 * {@attribute stereotype an optional stereotype}
111 * {@attribute ports a list of ports that are contained in
112 *   this interface}
113 */
114 /ArcInterface implements (Stereotype? "port")=> ArcElement =
115   Stereotype?
116   "port" ports:ArcPort ("," ports:ArcPort)* ";";
117
118 /**
119  * Used to embedd behavior implementations.
120  *
121  * {@attribute kind embedding kind}
122  * {@attribute behaviorEmbedding the concrete embedding}
123  */
124 ArcComponentImplementation implements (Stereotype?
125     "implementation" kind:Name Name? "{")=> ArcElement =
126   Stereotype? "implementation" kind:Name Name? "{"
127     BehaviorEmbedding(parameter kind)
128   "}";
129
130
131 /**
132 * An incoming port is used to receive messages, an
133 * outgoing port is used to send messages of a specific
```

```
134   * type.
135   *
136   * {@attribute stereotype an optional stereotype}
137   * {@attribute incoming true, if this is an incoming port}
138   * {@attribute outgoing true, if this is an outgoing port}
139   * {@attribute type the message type of this port}
140   * {@attribute name an optional name of this port}
141   */
142   /ArcPort =
143     Stereotype?
144     (incoming:["in"] | outgoing:["out"])
145     Type Name?;
146
147   /**
148   * A subcomponent is used to create one or more instances
149   *  of another component. This way the hierarchical
150   * structure of a component is defined.
151   *
152   * {@attribute stereotype an optional stereotype}
153   * {@attribute type the type of the instantiated component}
154   * {@attribute arguments list of configuration parameters
155   *   that are to be set, if the instantiated component is
156   *   configurable.}
157   * {@attribute instances list of instances that should be
158   *   created}
159   */
160   /ArcSubComponent implements
161       (Stereotype? "component" ReferenceType
162       ("(" | Name | ";" ))=> ArcElement =
163     Stereotype?
164     "component"
165     type:ReferenceType
166     ("(" arguments:CVExpression
167     ("," arguments:CVExpression)* ")" )?
168     (instances:ArcSubComponentInstance
169     ("," instances:ArcSubComponentInstance)* )? ";";
170
171   /**
172   * A subcomponent instance binds the name of an instance
173   * with an optional list of simple connectors used to
174   * connect this instance with other subcomponents/ports.
175   *
176   * {@attribute name the name of this instance}
177   * {@attribute connectors list of simple connectors}
178   */
179   /ArcSubComponentInstance =
180     Name
181     ("[" connectors:ArcSimpleConnector
182     (";" connectors:ArcSimpleConnector)* "]")?;
```

```
183
184   /**
185   * A connector connects one source port with one or many
186   * target ports.
187   *
188   * {@attribute source source port or component instance
189   *   name}
190   * {@attribute targets a list of target ports or component
191   *   instance names}
192   */
193   /ArcConnector implements
194       (Stereotype? "connect" QualifiedName "->")=>ArcElement=
195     Stereotype?
196     "connect" source:QualifiedName "->"
197     targets:QualifiedName ("," targets:QualifiedName)* ";";
198
199   /**
200   * A simple way to connect ports.
201   *
202   * {@attribute source the source port or component instance
203   *   name}
204   * {@attribute targets a list of target port or component
205   *   instance names}
206   */
207   /ArcSimpleConnector =
208     Stereotype? source:QualifiedName "->" targets:QualifiedName
209                           ("," targets:QualifiedName)*;
210
211   /**
212   * ArcParameters are used in configurable components.
213   *
214   * {@attribute Type the type of the parameter}
215   * {@attribute name the name of the parameter}
216   */
217   /ArcParameter =
218     Type Name;
219
220
221 /* ================================================*/
222 /* ================== ASTRULES ====================*/
223 /* ================================================*/
224   // replacement of ASTCNode with UMLPNode
225   ast ArcComponent astextends
226       /mc.umlp.common._ast.UMLPNode;
227   ast ArcComponentHead astextends
228       /mc.umlp.common._ast.UMLPNode;
229   ast ArcComponentBody astextends
230       /mc.umlp.common._ast.UMLPNode;
231   ast ArcPort astextends
```

```
232       /mc.umlp.common._ast.UMLPNode;
233   ast ArcConnector astextends
234       /mc.umlp.common._ast.UMLPNode;
235   ast ArcSimpleConnector astextends
236       /mc.umlp.common._ast.UMLPNode;
237   ast ArcSubComponent astextends
238       /mc.umlp.common._ast.UMLPNode;
239   ast ArcSubComponentInstance astextends
240       /mc.umlp.common._ast.UMLPNode;
241   ast ArcParameter astextends
242       /mc.umlp.common._ast.UMLPNode;
243 }
```

Listing C.1: `ArchitectureDiagram.mc`: Common MontiCore grammar for architectural diagrams.

C.2 MontiArc Grammar

MG

```
1  package mc.umlp.arc;
2
3  version "$LastChangedDate: 2015-01-19 12:45:03 +0100 (Mo, 19 Jan
4      2015) $, $LastChangedRevision: 3080 $";
5  /**
6   * Grammar for MontiArc. Extends common components with
7   * behavior information and configurations.
8   *
9   * @author Arne Haber
10  *         (last commit) $LastChangedBy: ahaber $
11  * @version $LastChangedDate: 2015-01-19 12:45:03 +0100 (Mo, 19
12  * Jan 2015) $
13  *          $LastChangedRevision: 3080 $
14  */
15 grammar MontiArc extends mc.umlp.arcd.ArchitectureDiagram {
16
17 /* ======================================================*/
18 /* ====================== OPTIONS =====================*/
19 /* ======================================================*/
20   options {
21     compilationunit ArcComponent
22     nostring
23     parser lookahead=5
24     lexer lookahead=7
25   }
26
27 /* ======================================================*/
28 /* ==================== PRODUCTIONS ====================*/
29 /* ======================================================*/
```

```
30 //
31
32   /**
33   * MontiArc components may contain arbitrary many
34   * configurations. These configurations have to
35   * implement this interface.
36   */
37   interface MontiArcConfig extends ArcElement;
38
39   /**
40   * An invariant constrains the behavior of a component.
41   *
42   * {@attribute kind the optional kind of this invariant.}
43   * {@attribute name name of the invariant}
44   * {@attribute invariantExpression the invariant defined
45   *   in the language 'kind'}
46   */
47   MontiArcInvariant implements
48       (Name? "inv" Name ":")=> ArcElement =
49     (kind:Name)? "inv" Name ":"
50     invariantExpression:InvariantContent(parameter kind) ";";
51
52   /**
53   * AutoConnect is used to connect ports automatically.
54   *
55   * {@attribute stereotype optional stereotype}
56   * {@attribute type autoconnect unambigous ports with the
57   *   same type}
58   * {@attribute port autoconnect unambigous ports with the
59   *   same name and compatible type}
60   * {@attribute off do not use autoconnection (default)}
61   */
62   MontiArcAutoConnect implements MontiArcConfig =
63     "autoconnect" Stereotype?
64     (["type"] | ["port"] | ["off"]) ";";
65
66   /**
67   * Autoinstantiate is used to instantiate inner components
68   * without generic parameters or configuration parameters
69   * automatically. If more then one instance of this inner
70   * component is created by using a reference, the
71   * automatically instantiated reference will disappear.
72   *
73   * {@attribute stereotype optional stereotype}
74   * {@attribute on turns autoinstantiate on}
75   * {@attribute off turns autoinstantiate off (default)}
76   */
77   MontiArcAutoInstantiate implements MontiArcConfig  =
78     "autoinstantiate" Stereotype?
```

```
79      (["on"] | ["off"]) ";";
80
81    /**
82    * Sets the timing of a component.
83    *
84    * {@attribute stereotype optional stereotype}
85    * {@attribute instant a timed component}
86    * {@attribute delayed a timed component with delay}
87    * {@attribute causalsync a causal synchronous component}
88    * {@attribute sync a synchronous component}
89    */
90    MontiArcTiming implements MontiArcConfig  =
91      "timing" Stereotype?
92      (["instant"] | ["delayed"] | ["untimed"] | ["causalsync"] |
93          ["sync"]) ";";
94
95  /* ========================================================*/
96  /* ================== ASTRULES ============================*/
97  /* ========================================================*/
98    // toString for ArcInvariant
99    ast MontiArcInvariant astextends
100       /mc.umlp.common._ast.UMLPNode =
101     method public String toString() {
102       return (this.getKind() != null ?
103           this.getKind() + " " : "") + "inv " +
104           this.getName();
105       };
106   // replacement of ASTCNode with UMLPNode
107   ast MontiArcTiming astextends
108       /mc.umlp.common._ast.UMLPNode;
109   ast MontiArcAutoInstantiate astextends
110       /mc.umlp.common._ast.UMLPNode;
111   ast MontiArcAutoConnect astextends
112       /mc.umlp.common._ast.UMLPNode;
113 }
```

Listing C.2: `MontiArc.mc`: MontiCore grammar for MontiArc.

C.3 I/O Test Language Grammar

MG

```
1  package de.monticore.lang.ctd;
2
3  version "$LastChangedDate: 2015-03-19 11:34:20 +0100 (Do, 19 Mrz
4      2015) $, $LastChangedRevision: 3135 $";
5  /**
6   * Grammar for intput/output component black box tests.
7   *
8   * @author Arne Haber
9   *         (last commit) $LastChangedBy: ahaber $
10  * @version $LastChangedDate: 2015-03-19 11:34:20 +0100 (Do, 19
11  * Mrz 2015) $
12  *          $LastChangedRevision: 3135 $
13  */
14 grammar ComponentIOTestDSL
15     extends de.monticore.lang.common.CommonValues {
16
17 /* ======================================================*/
18 /* ====================== OPTIONS =====================*/
19 /* ======================================================*/
20   options {
21     compilationunit ArcTestSuite
22     lexer lookahead=8
23   }
24
25 /* ======================================================*/
26 /* ==================== PRODUCTIONS ====================*/
27 /* ======================================================*/
28
29   /**
30    * Compilation unit of an I/O based test.
31    *
32    * {@attribute name name of the test suite}
33    * {@attribute componentUnderTest testee tested by the test
34    * suite}
35    * {@attribute elements contained test suite elements}
36    */
37   ArcTestSuite =
38     "testsuite" Name "for" ComponentUnderTest "{"
39         elements:ArcTestSuiteElement*
40     "}";
41
42   /**
43    * Testee that corresponds to a component instance.
44    *
45    * {@attribute type component type}
46    * {@attribute arguments optional configuration parameter
```

```
47    *            arguments}
48    */
49   ComponentUnderTest =
50     type:ReferenceType
51     ("(" arguments:CVExpression ("," arguments:CVExpression)* ")"
52         )?;
53
54   /**
55    * Interface for elements within a test suite.
56    */
57   interface ArcTestSuiteElement;
58
59   /**
60    * Interface for variable declarations.
61    */
62   interface ArcFieldDeclaration extends ArcTestSuiteElement;
63
64   /**
65   * Interface for stream variable declarations.
66   */
67   interface ArcStreamFieldDeclaration extends
68       ArcFieldDeclaration;
69
70   /**
71    * A field variable is visible within the complete
72    * test suite.
73    *
74    * {@attribute type field type}
75    * {@attribute name field name}
76    * {@attribute value optional value assignment}
77    */
78   ArcVariableField implements ArcFieldDeclaration =
79     Type Name ("=" Value)? ";";
80
81   /**
82    * A stream field is a stream that is visible within
83    * the complete test suite.
84    *
85    * {@attribute type type of the elements contained in the
86    *    stream}
87    * {@attribute name field name}
88    * {@attribute arcStream stream assigned to this variable }
89    */
90   /ArcStreamField implements ArcStreamFieldDeclaration =
91     "Stream" TypeArguments Name "=" ArcStream ";";
92
93   /ArcStreamMatcherField implements ArcStreamFieldDeclaration =
94     "StreamMatcher" TypeArguments Name "=" ArcStream ";";
95
```

```
96
97  /**
98   * Enumeration for setup options.
99   */
100 enum TestOption =
101   // run once before test suite
102   "@BeforeSuite" |
103   // run before each test
104   "@Before" |
105   // run after each test
106   "@After" |
107   // run once after test suite
108   "@AfterSuite";
109
110 /**
111  * Allows a specific test setup/tear down.
112  *
113  * {@attribute testOption defines when the setup
114  *    is executed}
115  * {@attribute setupEmbeddment what is to be executed}
116  */
117 / ArcTestSetup implements ArcTestSuiteElement =
118   TestOption "{"
119       SetupEmbeddment
120   "}";
121
122 /**
123  * External for setup/tearDown embedding.
124  */
125 external SetupEmbeddment;
126
127 /**
128  * A concrete test case.
129  *
130  * {@attribute name name of the test case}
131  * {@attribute repetitions optional amount of repetitions}
132  * {@attribute localSetup optional local test setup}
133  * {@attribute arcTestInput test input}
134  * {@attribute arcExpectedResult expected results}
135  * {@attribute arcExpectedStates expected states as String}
136  * {@attribute arcAssert optional assertions}
137  * {@attribute localTearDown optional local tear down}
138  */
139 / ArcTest implements ArcTestSuiteElement =
140   "test" Name ("repeat" repetitions:IntLiteral "times")? "{"
141       localVariables:ArcFieldDeclaration*
142       localSetup:ArcTestSetup?
143       ArcTestInput
144       ArcExpectedResult
```

```
145         ArcAssert?
146         localTearDown:ArcTestSetup?
147       "}";
148
149     /**
150      * Defines the input of a test case.
151      *
152      * {@attribute arcStreamAssignment input streams}
153      */
154     ArcTestInput =
155       "input" "{"
156           ArcStreamAssignment*
157       "}";
158
159     /**
160      * Defines the expected results of a test case.
161      *
162      * {@attribute arcStreamAssignment expected output streams}
163      */
164     ArcExpectedResult =
165       "expect" "{"
166           ArcStreamAssignment*
167       "}";
168
169     /**
170      * For further assertions.
171      *
172      * {@attribute assertDefinition contains the concrete
173      * assertions.}
174      */
175     ArcAssert =
176       "assert" "{"
177           AssertDefinition
178       "}";
179
180     /**
181      * External for assertions.
182      */
183     external AssertDefinition;
184
185     /**
186      * Assigns a stream to a port.
187      *
188      * {@attribute portName name of the port}
189      * {@attribute portNameAsString if the name of the port is
190      *     a keyword of this grammar, escape it as a string}
191      * {@attribute variable name of a stream field}
192      * {@attribute stream local definition of a stream}
193      */
```

```
194 ArcStreamAssignment =
195   (portName:Name | portNameAsString:String) ":"
196       (variable:Name | stream:ArcStream) ";";
197
198 /**
199  * Defines a stream.
200  *
201  * {@attribute elements elements of this stream.}
202  */
203 ArcStream =
204   "<" (elements:MultipliedStreamElement ("," 
205        elements:MultipliedStreamElement)* )? ">";
206
207 /**
208  * Element of a stream.
209  *
210  * {@attribute multiplier multiplier of the element}
211  * {@attribute negated optional, true, if this element is
212  * negated}
213  * {@attribute value value of the element}
214  * {@attribute optional optional, true, if this element is
215  * optional}
216  */
217 MultipliedStreamElement =
218   Multiplier ([negated:"!"])? Value ([optional:"?"])?;
219
220 /**
221  * Contains optional content. If set, it has
222  * either a fix amount or a range.
223  *
224  * {@attribute amount fix amount}
225  * {@attribute lower lower bound of a range}
226  * {@attribute upper upper bound of a range}
227  */
228 Multiplier =
229   ((amount:IntLiteral "*") |
230   ("[" lower:IntLiteral "," upper:IntLiteral "]" "*"))?;
231
232 /**
233  * A tick represents borders of time intervals.
234  */
235 Tick implements Value =
236   "Tk";
237
238 /**
239  * Any value.
240  */
241 UndefinedValue implements Value =
242   "_";
```

```
243
244   /**
245    * Bundles stream elements to groups.
246    *
247    * {@attribute elements list of aggregates elements}
248    */
249   ValueGroup implements ("(" MultipliedStreamElement)=> Value =
250     "(" elements:MultipliedStreamElement ("," 
251         elements:MultipliedStreamElement)* ")";
252
253 /* ======================================================*/
254 /* ================== ASTRULES ==========================*/
255 /* ======================================================*/
256   /**
257    * Adds a getName() and getType() method to field
258    * declarations.
259    */
260   ast ArcFieldDeclaration =
261     method public String getName(){}
262     method mc.types._ast.ASTType getType(){};
263
264   /**
265    * Adds a getTypeArguments() method.
266    */
267   ast ArcStreamFieldDeclaration =
268     method mc.types._ast.ASTTypeArguments getTypeArguments(){};
269
270   /**
271    * Redefines getPortName() method to either use attribute
272    * portName or use escaped portNameAsString.
273    */
274   ast ArcStreamAssignment =
275     method public String getPortName() {
276         if (this.portName == null && this.getPortNameAsString()
277                 != null) {
278             this.portName = this.getPortNameAsString().
279                 replace("\"", "");
280         }
281         return this.portName;
282     };
283
284   /**
285    * Adds a hasMultiplier() method to multiplied
286    * stream elements.
287    */
288   ast MultipliedStreamElement =
289     method public boolean hasMultiplier() {
290       return (multiplier.getAmount() != null ||
291         multiplier.getUpper() != null ||
```

```
292         multiplier.getLower() != null);
293     };
294
295   /**
296    * Adds a toString() method.
297    */
298   ast ArcStreamAssignment =
299     method public String toString() {
300         String res = "";
301         mc.ast.ASTNode parent = get_Parent().get_Parent();
302         if (parent instanceof ASTArcExpectedResult) {
303             res += "out";
304         }
305         else if (parent instanceof ASTArcTestInput) {
306             res += "in";
307         }
308         res += "port ";
309         res += getPortName();
310         return res;
311     };
312
313   /**
314    * Adds a toString() method.
315    */
316   ast ArcVariableField =
317     method public String toString() {
318         String res = getName();
319         res += " : ";
320         res += mc.types.helper.TypesPrinter.printType(getType());
321         return res;
322     };
323
324   /**
325    * Adds a toString() method.
326    */
327    ast ArcStreamField =
328     method public String toString() {
329         String res = getName();
330         res += " : ";
331         res +=
332             mc.types.helper.TypesPrinter.printType(getType());;
333         return res;
334     };
335   /**
336    * Adds a toString() method.
337    */
338    ast ArcStreamMatcherField =
339     method public String toString() {
340         String res = getName();
```

```
341         res += " : ";
342         res +=
343             mc.types.helper.TypesPrinter.printType(getType());;
344         return res;
345     };
346 }
```

Listing C.3: `ComponentIOTestDSL.mc`: MontiCore grammar for I/O tests.

C.4 Process Network Simulation Grammar

MG

```
1  grammar ProNetSim {
2
3    options {
4      compilationunit ProcessDefinition
5    }
6
7    interface ProcessDefinitionElement;
8
9    ProcessDefinition =
10     (["valueChain"] | ["processStep"]) Name "{"
11       ProcessDefinitionElement*
12     "}";
13
14   CommodityChannel implements ProcessDefinitionElement =
15     (["input"] | ["output"]) type:Name
16     ("[" amount:INT "]")?
17     receiver:Transport? ";";
18
19   ValueChainNode implements ProcessDefinitionElement =
20     (["valueChain"] | ["processStep"]) type:QualifiedName
21     Name? receiver:Transport? ";";
22
23   Transport =
24     "->" receivers:Target ("," receivers:Target)*;
25
26   Target =
27     targetNode:Name ("(" probability:INT ")")?;
28 }
```

Listing C.4: `ProNetSim.mc`: Essence of the ProNet^{sim} MontiCore grammar.

Appendix D

AADL Examples

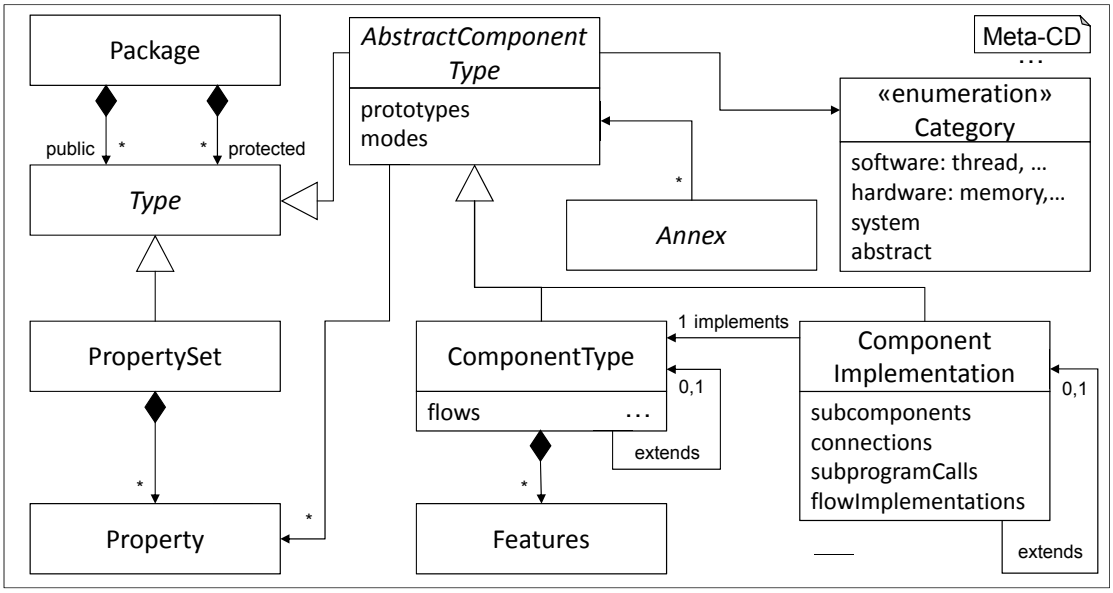


Figure D.1: Extract of the AADL metamodel (adapted from [FGH06, FG12]).

```aadl
package SubcompSnippets
public
  with Snippets_cfg;
  with Base_Types;
  with Data_Model;

  data Cmd
    properties
      Data_Model::Data_Representation => Enum;
      Data_Model::Enumerators=>("PULL", "PUSH");
      Data_Model::Representation => ("0", "1");
  end Cmd;

  abstract A
    features
      string: in event data port Base_Types::String;
      command: in event data port Cmd;
      integer: out event data port Base_Types::Integer;
  end A;
  abstract implementation A.AImpl
  -- ...
  end A.AImpl;

  abstract Ext extends A
  end Ext;
  abstract implementation Ext.ExtImpl extends A.AImpl
  end Ext.ExtImpl;

  abstract B
    features
      sIn: in event data port Base_Types::String;
      sOut: out event data port Base_Types::String;
    properties
      Snippets_cfg::p1=>1;
      Snippets_cfg::p2=>"bar";
  end B;
  abstract implementation B.BImpl
  end B.BImpl;

  abstract C
    prototypes
      K: data;
      V: data Base_Types::Integer;
    features
      msgIn: in event data port V;
      msgOut: out event data port K;
  end C;
  abstract implementation C.CImpl
```

```
49  end C.CImpl;
50
51  system D
52    features
53      sIn: in event data port Base_Types::String;
54      sOut: out event data port Base_Types::String;
55      iOut: out event data port Base_Types::Integer;
56  end D;
57
58  system implementation D.DImpl
59    subcomponents
60      a: abstract A.AImpl;
61      myExt: abstract Ext.ExtImpl;
62
63      myB1: abstract B.BImpl {
64        Snippets_cfg::p1=>5;
65        Snippets_cfg::p2=>"foo";
66      };
67
68      c: abstract C.CImpl (
69        K => data Base_Types::String,
70        V => data Base_Types::Integer
71      );
72
73    connections
74      cona: port sIn -> myB1.sIn;
75      conb: port c.msgOut -> sOut;
76      conc: port myB1.sOut -> a.string;
77      cond: port myB1.sOut -> myExt.string;
78      cone: port myExt.integer -> iOut;
79  end D.DImpl;
80
81 end SubcompSnippets;
```

Listing D.2: Complete AADL specifications with the examples from Section 3.6.

Appendix E

Tutorial Material

E.1 Implementations

Java

```
1  public class ABPInnerSenderImpl extends
2      abp.gen.AABPSender_ABPInnerSender {
3
4    /** ABP_S_1: To remember the last received ack. */
5    private boolean transmissionFlag;
6
7    /** ABP_S_2: To buffer messages. */
8    private Queue<String> buffer;
9
10   /** Used timeout. */
11   public static final int TIMEOUT = 3;
12   /** Default constructor. */
13   public ABPInnerSenderImpl() {
14     // ABP_S_3 implicitly:
15     // ABP_S_3.1 buffer empty -> RDY,
16     // ABP_S_3.2 buffer not empty -> W8ING
17     buffer = new LinkedList<String>(); // ABP_S_4
18     transmissionFlag = true;// ABP_S_1
19   }
20
21   @Override
22   protected void treatAck(Boolean ack) {
23     // message has been transmitted and the flag from the message
24     // has been returned
25     if (ack == transmissionFlag) { // ABP_S_6.2
26       // remove msg from buffer
27       buffer.poll();
28
29       // invert transmission flag
30       transmissionFlag = !transmissionFlag;
31
32       // if no further messages are buffered, stop timer
33       if (buffer.isEmpty()) {
```

```
34          // stop timer
35          sendSetTimer(-1);
36        }
37        // if messages are buffered, send next one
38        else {
39          sendMessage(buffer.peek());
40        }
41      }
42      // ABP_S_5.2: implicitly buffer is empty -> RDY -> ignore ack
43      // ABP_S_6.3: resend message, wrong flag received
44      else if (!buffer.isEmpty()) {
45        sendMessage(buffer.peek());
46      }
47    }
48
49    @Override
50    protected void treatMessage(String message) {
51      if (buffer.isEmpty()) { // ABP_S_5.1 RDY
52        // store into buffer, if we need to resend it
53        buffer.add(message);
54        sendMessage(message);
55      }
56      else { // ABP_S_6.1  W8ING
57        buffer.add(message);
58      }
59    }
60    /**
61     * Encapsulates given message into an {@link ABPMessage} that
62     * is
63     * emitted via port abpMessage. Additionally the timer is set
64     * and
65     * will call back in TIMEOUT time intervals.
66     *
67     * @param msg data to send
68     */
69    private void sendMessage(String msg) {
70      ABPMessage abpMessage = new ABPMessage(transmissionFlag,
71          msg);
72      sendAbpMessage(abpMessage);
73      sendSetTimer(TIMEOUT - 1);
74    }
75
76    @Override
77    protected void treatTimerEvent(Boolean message) {
78      // ABP_S_6.4: timer expired (message = true)
79      if (message && !buffer.isEmpty()) {
80        // resend message
81        sendMessage(buffer.peek());
82      }
```

```
83    }
84
85    @Override
86    protected void timeStep() {/*ignore*/}
87  }
```

Listing E.1: Initial behavior implementation of component `ABPInnerSender`.

```
1  public class ABPReceiverImpl extends abp.gen.AABPReceiver {
2
3    /** ABP_R_1: To store last received ack. */
4    private boolean lastReceivedMessageFlag;
5    /** Default constructor. */
6    public ABPReceiverImpl() {
7      lastReceivedMessageFlag = false; // ABP_R_2
8    }
9
10   @Override
11   protected void treatAbpMessage(ABPMessage message) {
12     // ABP_R_3: last and current message differ
13     if (lastReceivedMessageFlag != message.isAck()) {
14       // invert flag
15       lastReceivedMessageFlag = !lastReceivedMessageFlag;
16
17       // send transmitted message
18       sendMessage(message.getContent());
19
20       // send ack of received message
21       sendAck(lastReceivedMessageFlag);
22     }
23     else { // ABP_R_4
24       // send last ack to reorder the last message again
25       sendAck(lastReceivedMessageFlag);
26     }
27   }
28 }
```

Java

Listing E.2: Initial behavior implementation of component `ABPReceiver`.

E.2 I/O-Test Models

I/O-Test

```
1 package abp;
2
3 testsuite ABPSenderTest for ABPSender {
4
5   String msg1 = "Hello";
6   String msg2 = "MontiArc";
7   ABPMessage abpMsg1;
8   ABPMessage abpMsg2;
9
10  @Before {
11    abpMsg1 = new ABPMessage(true, msg1);
12    abpMsg2 = new ABPMessage(false, msg2);
13  }
14
15  test noInput {
16    input {
17      message : <10 * Tk>;
18      ack     : <10 * Tk>;
19    }
20    expect {
21      abpMessage : <10 * Tk>;
22    }
23  }
24
25  test encapsulateMsg {
26    input {
27      message : <msg1>;
28      ack : <>;
29    }
30    expect {
31      abpMessage : <abpMsg1>;
32    }
33  }
34
35  test repeatMessage {
36    input {
37      message : <msg1, 15 * Tk>;
38      ack     : <15 * Tk>;
39    }
40    expect {
41      abpMessage : <5 * (abpMsg1, 3 * Tk), abpMsg1>;
42    }
43  }
44
45  test alternateMessages {
46    input {
```

```
47        message : <5 * (msg1, Tk, msg2, Tk)>;
48        ack     : <5 * (Tk, true, Tk, false)>;
49      }
50      expect {
51        abpMessage : <5 * (abpMsg1, Tk, abpMsg2, Tk)>;
52      }
53    }
54
55    test bufferMessages {
56      input {
57        message : <5 * (msg1, msg2), 10 * Tk>;
58        ack     : <5 * (Tk, true, Tk, false)>;
59      }
60      expect {
61        abpMessage : <5 * (abpMsg1, Tk, abpMsg2, Tk)>;
62      }
63    }
64  }
```

Listing E.3: I/O-Test suite for component `ABPSender`.

I/O-Test

```
1  package abp;
2  import abp.ABPReceiver;
3
4  testsuite ABPReceiverTest for ABPReceiver {
5
6    ABPMessage ack;
7    ABPMessage nack;
8
9    @Before {
10     ack = new ABPMessage(true, "ACK");
11     nack = new ABPMessage(false, "NACK");
12   }
13
14   test noMessages {
15     input {
16       // type: ABPMessage
17       abpMessage : <100 * Tk>;
18     }
19     expect {
20       // type: java.lang.Boolean
21       ack : <100 * Tk>;
22       // type: java.lang.String
23       message : <100 * Tk>;
24     }
25   }
26
27   test testRepeatAckTimed {
28     input {
```

```
29        // type: abp.ABPMessage
30        abpMessage : <10 * (ack, Tk)>;
31      }
32      expect {
33        // type: java.lang.Boolean
34        ack : <10 * (true, Tk)>;
35        // type: java.lang.String
36        message : <"ACK", 10 * Tk>;
37      }
38    }
39
40    test testRepeatAckUntimed {
41      input {
42        // type: abp.ABPMessage
43        abpMessage : <10 * ack>;
44      }
45      expect {
46        // type: java.lang.Boolean
47        ack : <10 * true>;
48        // type: java.lang.String
49        message : <"ACK">;
50      }
51    }
52
53    test testRepeatNackTimed {
54      input {
55        // type: abp.ABPMessage
56        abpMessage : <10 * (nack, Tk)>;
57      }
58      expect {
59        // type: java.lang.Boolean
60        ack : <10 * (false, Tk)>;
61        // type: java.lang.String
62        message : <10 * Tk>;
63      }
64    }
65
66    test testRepeatNackUntimed {
67      input {
68        // type: abp.ABPMessage
69        abpMessage : <10 * nack>;
70      }
71      expect {
72        // type: java.lang.Boolean
73        ack : <10 * false>;
74        // type: java.lang.String
75        message : <>;
76      }
77    }
```

```
78
79   test testRepeatAckNackTimed {
80     input {
81       // type: abp.ABPMessage
82       abpMessage : <10 * (ack, Tk, nack, Tk)>;
83     }
84     expect {
85       // type: java.lang.Boolean
86       ack : <10 * (true, Tk, false, Tk)>;
87       // type: java.lang.String
88       message : <10 * ("ACK", Tk, "NACK", Tk)>;
89     }
90   }
91
92   test testRepeatAckNackUntimed {
93     input {
94       // type: abp.ABPMessage
95       abpMessage : <10 * (ack, nack)>;
96     }
97     expect {
98       // type: java.lang.Boolean
99       ack : <10 * (true, false)>;
100      // type: java.lang.String
101      message : <10 * ("ACK", "NACK")>;
102    }
103  }
104 }
```

Listing E.4: I/O-Test suite for component `ABPReceiver`.

I/O-Test

```
1  package abp;
2
3  testsuite ABPTest for ABP {
4
5    test noInput {
6      input {
7        // type: java.lang.String
8        msg : <10 * Tk>;
9      }
10     expect {
11       // type: java.lang.String
12       transmittedMsg : <11 * Tk>;
13     }
14   }
15
16   String msg = "Hello MontiArc";
17
18   test messageTransmission repeat 10000 times {
19     input {
```

```
20        // type: java.lang.String
21        msg : <msg, 100 * Tk>;
22      }
23      expect {
24        // type: java.lang.String
25        transmittedMsg : <[1, 101] * Tk, msg, [0, 101] * Tk>;
26      }
27    }
28 }
```

Listing E.5: I/O-Test suite for system component ABP.

E.3 White-Box Tests

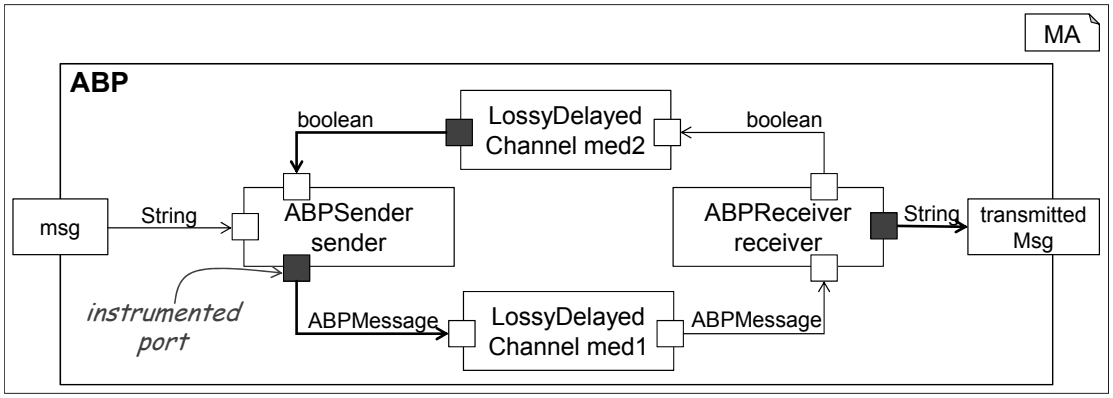


Figure E.6: Instrumented ports of ABP's subcomponents in the white-box test given in Listing E.7.

```
1 public class ABPWhiteBoxTestWithParameterManipulation {
2   private final Tick<String> tick = Tick.<String> get();
3   private ABP testee;
4   private IStream<ABPMessage> senderOutStream;
5   private IStream<Boolean> senderAckInStream;
6   private IStream<String> receiverOutStream;
7
8   @Before
9   public void setUp() {
10    setUpFactory();
11    testee = new ABP();
12    IScheduler s = SchedulerFactory.createDefaultScheduler();
13    s.setPortFactory(new TestPortFactory());
14    testee.setup(s, new SimpleErrorHandler());
15
16    senderOutStream = ((ITestPort<ABPMessage>) testee.getSender()
```

JUnit

```
17          .getAbpMessage()).getStream();
18      receiverOutStream = ((ITestPort<String>) testee.getReceiver()
19          .getMessage()).getStream();
20      senderAckInStream = ((ITestPort<Boolean>) testee.getMed2()
21          .getPortOut()).getStream();
22    }
23    private void setUpFactory() {
24      // register test factory
25      LossyDelayedChannelFactory.register(
26          new LossyDelayedChannelTestFactory());
27    }
28    @After
29    public void tearDown() {
30      LossyDelayedChannelFactory.reset();
31    }
32
33    @Test
34    public void testAckOfMessage() {
35      String m = "Hello MontiArc";
36      boolean expectedAck = true;
37      testee.getMsg().accept(m);
38      testee.getMsg().accept(tick);
39      testee.getMsg().accept(tick);
40
41      // message sent once?
42      assertEquals(1, senderOutStream.getUntimedHistory().size());
43      assertEquals(expectedAck,
44          senderOutStream.getUntimedHistory().get(0).isAck());
45      assertEquals(m,
46          senderOutStream.getUntimedHistory().get(0).getContent());
47
48      // Message transmitted once in time interval 1?
49      assertEquals(1,
50          receiverOutStream.getUntimedHistory().size());
51      assertTrue(receiverOutStream.getUntimedHistory().contains(m));
52      assertEquals(1, receiverOutStream.firstTimeIntervalOf(m));
53
54      // Message acknowledged in time interval 2?
55      assertEquals(1,
56          senderAckInStream.getUntimedHistory().size());
57      assertTrue(senderAckInStream.getUntimedHistory().get(0));
58      assertEquals(2, senderAckInStream.firstTimeIntervalOf(true));
59
60      // Two time frames have passed?
61      assertEquals(2, testee.getLocalTime());
62    }
63
64    class LossyDelayedChannelTestFactory extends
65        LossyDelayedChannelFactory {
```

```
66      @Override
67      protected <T> ILossyDelayedChannel<T> doCreate(
68          IRandomFunction f, int delay) {
69        // always set loss rate to zero
70        return super.doCreate(new ControlledRandom("1"), delay);
71      }
72    }
73  }
```

Listing E.7: White-box test that uses parameter manipulation to adjust the configuration of ABP's LossyChannel subcomponents.

JUnit

```
1  public class ABPMockingTest {
2    private ABP testee;
3    private TestPort<String> receiverOut;
4
5    protected IScheduler getScheduler() {
6      return SchedulerFactory.createDefaultScheduler();
7    }
8    @Before
9    public void setUp() {
10     // init user defined factory
11     LossyDelayedChannelFactory.register(
12         new LossyDelayedChannelTestFactory());
13     testee = new ABP();
14     testee.setup(getScheduler(), new SimpleErrorHandler());
15     receiverOut = new TestPort<String>();
16     testee.setTransmittedMsg(receiverOut);
17   }
18
19   @After
20   public void tearDown() {
21     LossyDelayedChannelFactory.reset();
22   }
23
24   @Test
25   public void testAlternatingBit() {
26     LDCMock<ABPMessage> m1 = (LDCMock<ABPMessage>)
27         testee.getMed1();
28     LDCMock<Boolean> m2 = (LDCMock<Boolean>) testee.getMed2();
29     int amount = 1000;
30     for (int i = 0; i < amount; i++) {
31       String msg = "Msg " + i;
32       testee.getMsg().accept(msg);
33       assertEquals(i + 1, m1.called);
34       assertEquals(i + 1, m2.called);
35       assertEquals(msg, m1.lastMessage.getContent());
36       if (i % 2 == 0) {
37         assertTrue(m1.lastMessage.isAck());
```

```
38          assertTrue(m2.lastMessage);
39        }
40        else {
41          assertFalse(m1.lastMessage.isAck());
42          assertFalse(m2.lastMessage);
43        }
44      }
45      assertEquals(amount, receiverOut.getStream().size());
46    }
47
48    class LossyDelayedChannelTestFactory
49        extends LossyDelayedChannelFactory {
50      /**
51       * @see ma.sim.gen.factories.LossyDelayedChannelFactory
52       *   #doCreate(int, int)
53       */
54      @Override
55      protected <T> ILossyDelayedChannel<T> doCreate(IRandomFunction
56          f,
57          int delay) {
58        return new LDCMock<T>(new ControlledRandom("1"), delay);
59      }
60    }
61
62    class LDCMock<T> extends ALossyDelayedChannelStub<T> {
63      public int called = 0;
64      public T lastMessage;
65
66      public LDCMock(IRandomFunction f, int delay) {
67        super(f, delay);
68      }
69
70      /**
71       * @see ma.sim.gen.ALossyDelayedChannelStub#treatPortIn(
72       *   java.lang.Object)
73       */
74      @Override
75      protected void treatPortIn(T message) {
76        sendPortOut(message);
77        called++;
78        lastMessage = message;
79      }
80    }
81  }
```

Listing E.8: Mocking subcomponents in a test.

E.4 Generalized Components

MA

```
1 package abp;
2
3 component ABPReceiver<T> {
4   timing untimed;
5   port
6     in ABPMessage<T> abpMessage,
7     out Boolean ack,
8     out T message;
9 }
```

Listing E.9: Generalized receiver component model.

MA

```
1 package abp;
2
3 import ma.sim.LossyDelayedChannel;
4 import ma.sim.IRandomFunction;
5
6 component ABP<T>[IRandomFunction rand, int delay, int timeout] {
7
8   port
9     in  T msg,
10    out T transmittedMsg;
11
12  component ABPSender<T>(timeout)
13    sender [abpMessage -> med1.portIn];
14
15  component LossyDelayedChannel<ABPMessage<T>>(rand, delay)
16    med1 [portOut -> receiver.abpMessage];
17
18  component ABPReceiver<T>
19     receiver [ack -> med2.portIn;
20               message -> transmittedMsg];
21
22  component LossyDelayedChannel<Boolean>(rand, delay)
23    med2 [portOut -> sender.ack];
24
25  connect msg -> sender.message;
26 }
```

Listing E.10: Generalized abp component model.

E.5 Optimization Testing

Java

```
public class RunAbpStatistics {
  public static void main(String[] args) {
    // create a new workbook
    Workbook wb = new HSSFWorkbook();
    // create a new sheet
    Sheet sh = wb.createSheet();
    int rownum = 0;
    createHeader(sh);

    final int experimentAmount = 5000;
    final Tick<String> tick = Tick.<String> get();
    final String transmittedMsg = "Hello MontiArc";

    for (int lossrate = 10; lossrate <= 80; lossrate += 10) {
      for (int delay = 1; delay <= 10; delay++) {
        int timeout = 2 * delay + 1;
        // from apache commons math
        final DescriptiveStatistics ds = new
            DescriptiveStatistics();
        for (int i = 0; i < experimentAmount; i++) {
          // setup ABP
          ABP<String> abp = new ABP<String>(new
              JavaRandom(lossrate), delay, timeout);
          IScheduler s =
              SchedulerFactory.createDefaultScheduler();
          s.setPortFactory(new TestPortFactory());
          abp.setup(s, new SimpleErrorHandler());
          TestPort<Boolean> ack = (TestPort<Boolean>)
              abp.getSender().getAck();

          // sent message
          abp.getMsg().accept(transmittedMsg);
          // sent ticks until msg is acknowledged
          while (ack.getStream().getUntimedHistory().isEmpty()) {
            abp.getMsg().accept(tick);
          }
          // passed simulation time
          int time = ack.getStream().getCurrentTime();
          ds.addValue(time);
        }
        System.out.println("Computed statistics for ABP(" +
            lossrate + ", " + delay + ", " + timeout + ")");
        rownum++;
        addToTable(sh, rownum, lossrate, delay, timeout,
            ds.getMin(), ds.getMax(), ds.getMean(),
            ds.getStandardDeviation(), ds.getPercentile(50));
```

```
47        }
48      }
49      writeTable(wb, "target/statistics/abp.xls");
50    }
51
52    /**
53     * Stores the given {@link Workbook} in a file determined by the
54     *
55     * given location.
56     *
57     * @param wb {@link Workbook} to store
58     * @param location file location
59     */
60    static void writeTable(Workbook wb, String location) {
61      FileOutputStream out;
62      try {
63        File result = new File(location);
64        if (!result.exists()) {
65          result.getParentFile().mkdirs();
66          result.createNewFile();
67        }
68        out = new FileOutputStream(result);
69        wb.write(out);
70        out.close();
71        System.out.println("Results written to: " +
72            result.getAbsolutePath());
73      }
74      catch (FileNotFoundException e) {
75        e.printStackTrace();
76      }
77      catch (IOException e) {
78        e.printStackTrace();
79      }
80    }
81
82    /**
83     *
84     * Creates a table row with the given value entries.
85     *
86     * @param sh current sheet
87     * @param rownum row number to create
88     * @param lossrate lossrate value
89     * @param delay delay value
90     * @param timeout timeout value
91     * @param min min value
92     * @param max max value
93     * @param mean mean value
94     * @param stdDev standard deviation value
95     * @param median median value
```

```
96      */
97     static void addToTable(Sheet sh, int rownum, double lossrate,
98           double delay, double timeout, double min, double max,
99           double mean, double stdDev, double median) {
100      Row r = sh.createRow(rownum);
101      r.createCell(0).setCellValue(lossrate);
102      r.createCell(1).setCellValue(delay);
103      r.createCell(2).setCellValue(timeout);
104      r.createCell(3).setCellValue(min);
105      r.createCell(4).setCellValue(max);
106      r.createCell(5).setCellValue(mean);
107      r.createCell(6).setCellValue(stdDev);
108      r.createCell(7).setCellValue(median);
109    }
110
111    /**
112     *
113     * Creates a table header row in the given {@link Sheet}.
114     *
115     * @param sh {@link Sheet} to use
116     */
117    static void createHeader(Sheet sh) {
118      Row r = sh.createRow(0);
119      r.createCell(0).setCellValue("LossRate");
120      r.createCell(1).setCellValue("Delay");
121      r.createCell(2).setCellValue("Timeout");
122      r.createCell(3).setCellValue("Min");
123      r.createCell(4).setCellValue("Max");
124      r.createCell(5).setCellValue("Mean");
125      r.createCell(6).setCellValue("Standard Deviation");
126      r.createCell(7).setCellValue("Median");
127    }
128 }
```

Listing E.11: Collecting statistical data over the impact of the parameters loss rate and delay of component `ABP`.

Java

```
1 public class RunAbpStatisticsParallel {
2
3   private final static int EXPERIMENT_AMOUNT = 5000;
4
5   /**
6    * Used to store simulation parameters and results.
7    */
8   static class SimResult {
9     private final int lossrate, delay, timeout;
10    private final DescriptiveStatistics stats;
11
12    private SimResult(int lossrate, int delay, int timeout) {
13      this.lossrate = lossrate;
14      this.delay = delay;
15      this.timeout = timeout;
16      this.stats = new DescriptiveStatistics();
17    }
18    private void addResult(double time) {
19      stats.addValue(time);
20      int stepSize = 20;
21      if (lossrate > 60) {
22        stepSize = 5;
23      }
24      int proc = EXPERIMENT_AMOUNT / 100 * stepSize;
25      int amount = stats.getValues().length;
26      if (amount % proc == 0) {
27        System.out.println((amount / proc * stepSize)  +
28            " % finished of (" + lossrate + ", " +
29            delay + ", " + timeout + ")");
30      }
31    }
32  }
33
34  /**
35   * Executes a concrete scenario and stores the result using
36   * {@link SimResult#addResult(double)}.
37   */
38  static class SimulationTask implements Runnable {
39    private final CountDownLatch latch;
40    private final SimResult result;
41
42    private SimulationTask(CountDownLatch l, SimResult result) {
43      this.latch = l;
44      this.result = result;
45    }
46
47    @Override
48    public void run() {
```

```
49      String transmittedMsg = "Hello MontiArc";
50      Tick<String> tick = Tick.<String> get();
51      // setup ABP
52      ABP<String> abp = new ABP<String>(
53          new JavaRandom(result.lossrate), result.delay,
54          result.timeout);
55      IScheduler s = SchedulerFactory.createDefaultScheduler();
56      s.setPortFactory(new TestPortFactory());
57      abp.setup(s, new SimpleErrorHandler());
58      TestPort<Boolean> ack = (TestPort<Boolean>)
59          abp.getSender().getAck();
60
61      // sent message
62      abp.getMsg().accept(transmittedMsg);
63
64      // sent ticks until msg is acknowledged
65      while (ack.getStream().getUntimedHistory().isEmpty()) {
66        abp.getMsg().accept(tick);
67      }
68      // passed simulation time
69      int time = ack.getStream().getCurrentTime();
70      result.addResult(time);
71      latch.countDown();
72    }
73  }
74
75  public static void main(String[] args) throws
76      InterruptedException {
77    int cores = Runtime.getRuntime().availableProcessors();
78    ExecutorService exec = Executors.newFixedThreadPool(cores);
79    final int lossRateMax = 80;
80    final int delayMax = 10;
81    // used to monitor the amount of finished simulations
82    final CountDownLatch latch = new CountDownLatch(
83        (lossRateMax / 10) * delayMax * EXPERIMENT_AMOUNT);
84
85    List<SimResult> results = Lists.newLinkedList();
86    for (int lossr = 10; lossr <= lossRateMax; lossr += 10) {
87      for (int delay = 1; delay <= delayMax; delay++) {
88        int timeout = 2 * delay + 1;
89        SimResult r = new SimResult(lossr, delay, timeout);
90        results.add(r); // store result object
91        for (int exp = 0; exp < EXPERIMENT_AMOUNT; exp++) {
92          exec.submit(new SimulationTask(latch, r));
93        }
94      }
95    }
96    latch.await(); // wait until all tasks have finished
97    // export results to spread sheet...
```

```
98      // create a new workbook
99      Workbook wb = new HSSFWorkbook();
100     // create a new sheet
101     Sheet sh = wb.createSheet();
102     createHeader(sh);
103     int rownum = 1;
104     for (SimResult r : results) {
105       DescriptiveStatistics ds = r.stats;
106       addToTable(sh, rownum, r.lossrate, r.delay, r.timeout,
107           ds.getMin(), ds.getMax(), ds.getMean(),
108           ds.getStandardDeviation(), ds.getPercentile(50));
109       rownum++;
110     }
111     writeTable(wb, "target/statistics/abp_parallel.xls");
112     exec.shutdown();
113   }
114 }
```

Listing E.12: Parallel execution of simulations to collext statistical data over the impact of the parameters loss rate and delay of component `ABP`.

E.6 Distributed Simulation

Java

```
public final class RemoteAbpSender implements Runnable {

  private IDelayedSender<String> sender;
  private static final Tick<String> TICK = Tick.<String> get();

  /**
   * Configures a DelayedSender for a distributed simulation.
   * @param addr IP address of the ABP receiver.
   */
  public void setUp(String addr) {
    sender = DelayedSenderFactory.create(2, 1);

    // configure scheduler to use TCPPortFactory
    IScheduler sched = SchedulerFactory.createDefaultScheduler();
    IPortFactory factory = new TCPPortFactory();
    sched.setPortFactory(factory);
    sender.setup(sched, new SimpleErrorHandler());

    // configure abp out port
    TCPPort<ABPMessage<String>> abpOut =
        new TCPPort<ABPMessage<String>>();
    abpOut.addReceiver(addr, PORT_ABP);
    sender.setAbpMessage(abpOut);

    // Get and configure ack in port
    IInTCPPort<Boolean> ack = (IInTCPPort<Boolean>)
        sender.getAck();
    ack.startListenOn(PORT_ACK);
  }

  /**
   * Sends the given message to port message.
   * @param msg message to send
   */
  public void sendMessage(String msg) {
    System.out.println("Sending: " + msg);
    sender.getMessage().accept(msg);
  }

  /**
   * Sends a tick to port message.
   */
  public void sendTick() {
    sender.getMessage().accept(TICK);
  }

```

```
47   /**
48    * Main method to start a distributed sender.
49    *
50    * @param args not used
51    */
52   public static void main(String[] args) {
53       String addr;
54       if (args.length >= 2) {
55           addr = args[1];
56       }
57       else {
58           addr = RECEIVER_HOST;
59       }
60     final RemoteAbpSender ras = new RemoteAbpSender();
61     ras.setUp(addr);
62     ras.run();
63   }
64
65   @Override
66   public void run() {
67     // periodically triggers ras to send a tick
68     Ticker ticker = new Ticker(this);
69     new Thread(ticker).start();
70
71     // scanner is used to read input from console
72     Scanner scanner = new Scanner(System.in);
73     try {
74         boolean running = true;
75         while (running) {
76             String input = scanner.next();
77             if ("exit".equals(input)) {
78                 running = false;
79             }
80             this.sendMessage(input);
81         }
82     }
83     finally {
84         // close scanner and stop threads
85         scanner.close();
86         ticker.stop();
87         ((TCPPort<ABPMessage<String>>)
88             sender.getAbpMessage()).stop();
89         ((TCPPort<Boolean>) sender.getAck()).stop();
90         ((TCPPort<String>) sender.getMessage()).stop();
91     }
92   }
93 }
```

Listing E.13: Configuration and execution of a distributed ABP sender.

Appendix F

Language Extension Material

```java
@Override
protected void bindMontiArcModelingLanguage() {
  // left empty...we simply override default binding.
}
@Inject @Provides
ModelingLanguage getExtendedMontiArcLanguage(ILanguage component)
    {
  MontiArcLanguage lng = new MontiArcLanguage(component) {
    @Override
    public DSLRootFactory<?> getRootFactory(
        IModelInfrastructureProvider ip, IErrorDelegator eh,
            String enc) {
      return new ExtendedMontiArcRootFactory(ip, eh, enc);
    }
  };
  lng.getDslRootClassForUserNames().put(MONTI_ARC_ROOT_NAME,
      ExtendedMontiArcRoot.class);
  lng.setRootClass(ExtendedMontiArcRoot.class);
  lng.addExecutionUnit(WF_PARSE, new
      ExtendedMontiArcParsingWorkflow());
  lng.addExecutionUnit(WF_CREATE_SYMTAB,
      new CreateExportedInterfaceWorkflow<ExtendedMontiArcRoot>(
      ExtendedMontiArcRoot.class, lng));
  lng.addExecutionUnit(WF_INIT_CHECK,
      new PrepareCheckWorkflow<ExtendedMontiArcRoot>(
              ExtendedMontiArcRoot.class, lng));
  lng.addExecutionUnit(WF_PRE_CHECK_TRAFO,
      new PreCoCoCheckMontiArcTransformationWorkflow
      <ExtendedMontiArcRoot>(ExtendedMontiArcRoot.class));
  return lng;
}
```

Listing F.1: Method `getExtendedMontiArcLanguage(...)` which provides an extended MontiArc modeling language in a guice module that extends the MontiArc default module.

```java
public class PortCountWorkflow<T extends DSLRoot<? extends
    ASTMCCompilationUnit>> extends DSLWorkflow<T> {

  public static PortCountWorkflow<MontiArcRoot> create() {
      return new
          PortCountWorkflow<MontiArcRoot>(MontiArcRoot.class);
  }
  public PortCountWorkflow(Class<T> rootClass) {
    super(rootClass);
  }

  @Override
  public void run(T dslroot) {
    ASTArcComponent component = dslroot.getAst().getType();
    int ports = countPorts(component);

    StringBuilder sb = new StringBuilder();
    sb.append("Component ");
    sb.append(component.getName());
    sb.append(" has ");
    sb.append(ports);
    sb.append(" ports.");
    GeneratedFile result =
        GeneratedFileHelper.createDeferredFileFromQualifiedName(
        dslroot.getName(), "stats");
    result.setContent(sb);
    dslroot.addFile(result);
  }

  private int countPorts(ASTArcComponent component) {
    int result = 0;
    for (ASTArcElement element :
        component.getBody().getArcElement()) {
      if (element instanceof ASTArcInterface) {
        result += ((ASTArcInterface) element).getPorts().size();
      }
      else if (element instanceof ASTArcComponent) {
        result += countPorts(((ASTArcComponent) element));
      }
    }
    return result;
  }
}
```

Listing F.2: A custom workflow that counts the ports of processed components.

```java
public class CustomTrafoConfigFactory extends
    MontiArcTrafoConfigurationFactory {

  @Override
  public Optional<ITrafoConfiguration> create(final
      SymbolTableInterface symtab, String workflow) {
    // get MontiArc transformations from super class for the given
    // workflow
    final Optional<ITrafoConfiguration> maTrafos =
        super.create(symtab, workflow);
    ITrafoConfiguration cfg = null;
    // add a transformation that is executed before context
    // condition checks
    if (workflow.equals(ToolConstants.WF_PRE_CHECK_TRAFO)) {
      cfg = new ITrafoConfiguration() {
        private final List<ITransformator> trafos =
            Lists.newArrayList();
        @Override
        public List<ITransformator> getTrafos() {
          if (trafos.isEmpty()) {
            // add custom transformation
            trafos.add(new ExpandSubcomponentTrafo(symtab));
            // add MontiArc transformations, if present
            if (maTrafos.isPresent()) {
              trafos.addAll(maTrafos.get().getTrafos());
            }
          }
          return trafos;
        }
      };
    }
    if (cfg != null) {
      return Optional.of(cfg);
    }
    else if (maTrafos.isPresent()) {
      return maTrafos;
    }
    // no transformation from this factory or a superclass
    else {
      return Optional.absent();
    }
  }
}
```

Listing F.3: A custom transformation configuration factory that adds an additional transformation.

Transformation Interface	Description
ICompilationUnitTransformation	Provides methods to transform the compilation unit at start and end of the abstract syntax tree (AST) traversal.
IComponentTransformation	Provides methods to transform component nodes (see Listing 3.18 on page 53). These methods are called before and after traversing a component AST node.
IComponentHeadTransformation	Provides a method to transform the head of a component (see Listing 3.19 on page 53). It may be used to transform generic type parameters, configuration parameters, and the extends clause of the current component.
IPortTransformation	Provides a method to transform ports of a component (see Listing 3.21 on page 54).
IComponentImplementation-Transformation	Provides a method to transform component implementations (see Listing 3.25 on page 56). A IComponentImplementationTransformation is responsible for a certain embedded implementation language.
ISubComponentTransformation	Provides a method to transform subcomponents. Please note that this transformation is called on subcomponent AST nodes only (see Listing 3.22 on page 55). Available subcomponent instance nodes (see Listing 3.23) have to be handled by this transformation manually.
IConnectorTransformation	Provides methods to transform simple (see Listing 3.23 on page 55) and normal connectors (see Listing 3.24).
IParameterListTransformation	Provides a method to transform the parameter list of a parameterizable component (see Listing 3.19 on page 53).
ITypeParametersTransformation	Provides a method to transform generic type parameters of a generic component (see Listing 3.19 on page 53, l. 2).
IAutoConnectTransformation	Provides a method to transform autoconnect statements (see Listing 3.27 on page 57).
IAutoInstantiateTransformation	Provides a method to transform autoinstantiate statements (see Listing 3.28 on page 57).

Table F.4 continued on next page

Transformation Interface	Description
IConstraintTransformation	Provides a method to transform constraints (see Listing 3.26 on page 56).
ITimingTransformator	Provides a method to transform the timing of a component (see Listing 3.29 on page 57).

Table F.4: Available transformation interfaces provided by MontiArc's transformation framework for the ArcD (top) and MontiArc language (bottom).

Hook Point	AST Node	Location	Description
arcElementHook	ArcElement	class body	Generic extension points for elements of a component.
checkConstraints-Hook	ArcComponent	checkConstraints()	Extension points for constraint definitions.
constructorHook	ArcComponent	constructor	Extension point to add code at the end of the generated constructor (cf. Figure 5.31 on page 153).
getLocalTime-Hook	ArcComponent	getLocalTime()	Extension point to add code to the `getLocalTime()` method of decomposed components.
handleMessage-Hook	ArcComponent	handleMessage()	Allows to add code to the end of the `handleMessage(...)` method. Called for timed, timed delaying, and untimed components (cf. Figure 5.33 on page 154).
handleMessage-TimeSyncHook	ArcComponent	handleMessage()	Allows to add code to the end of the `handleMessage(...)` method. Called for time-synchronous components (cf. Figure 5.33 on page 154).
handleTickStart-Hook	ArcComponent	handleTick()	Extension point to add code at the start of the `handleTick()` method (cf. Figure 5.35 on page 156).
handleTickEnd-Hook	ArcComponent	handleTick()	Extension point to add code at the end of the `handleTick()` method (cf. Figure 5.35 on page 156).

Table F.5 continued on next page

Hook Point	AST Node	Location	Description
newMethods-Hook	ArcComponent	class body	Generic extension point to add new methods to the generated class.
setupStartHook	ArcComponent	setup()	Extension point that allows to add code to the start of the setup() method (cf. Figure 5.32 on page 153 for atomic components, cf. Section 5.4.3 on page 156 for decomposed components).
setupEndHook	ArcComponent	setup()	Extension points that allows to add code to the end of the setup() method (cf. setupStartHook).
getPortHook	ArcPort	port getter	Allows to add code to the getter methods of a component (see Section 5.4.2 on page 151 and Section 5.4.3 on page 157 for a description of the generated code for ports of atomic and decomposed component).
setPortHook	ArcPort	port setter	Allows to add code to the outgoing port setter methods of a component (cf. hook point getPortHook).
sendOutPortHook	ArcPort	send methods	Allows to add code to the send methods that are generated for each outgoing port of an atomic component (cf. Figure 5.34 on page 155).
getSubcomponent-Hook	ArcSubComponentInstance	subcomponent getter	Extension point that adds code to the end of the generated protected getter method for subcomponents (cf. Figure 5.37 on page 158).
getConfigParam-Hook	ArcParameter	parameter getter	Allows to add code to the end of the protected parameter getter methods that are generated for configurable components (cf. Figure 5.31 on page 153).

Table F.5: Unbound hook points that serve as extension points of the MontiArc component generator. These extensions are called with the given AST node and generate code within the given location of the target code.

Appendix G

Curriculum Vitae

Name	Haber
Vorname	Arne
Geburtstag	20.03.1983
Geburtsort	Wolfsburg
Staatsangehörigkeit	deutsch
seit 2015	System Analytiker & Software Entwickler Schier Consult GmbH Braunschweig
2009 - 2014	Wissenschaftlicher Mitarbeiter Lehrstuhl für Software Engineering RWTH Aachen
2009	Abschluss als Diplom-Wirtschaftsinformatiker
2003 - 2009	Studium der Wirtschaftsinformatik an der TU Braunschweig
2002 - 2003	Zivildienst
2002	Abitur
1993 - 2002	Orientierungsstufe & Gymnasium Wolfsburg
1989 - 1993	Grundschule Wendschott

Bibliography

[AAAG+05] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, and Tony Tseng. Software Architecture with Acme and ArchJava (Research Demonstration). In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MS, May 2005.

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 18(5):9–20, 1993.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[ACN02b] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE) 2002*. ACM Press, 2002.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[AGST04] Jonathan Aldrich, David Garlan, Bradley Schmerl, and Tony Tseng. Modeling and Implementing Software Architecture with Acme and ArchJava. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 6–7, New York, NY, USA, 2004. ACM.

[AJ002] ArchJava Language Reference Manual, Version 1.0, 2002. `http://archjava.fluid.cs.cmu.edu/papers/archjava-language.pdf`.

[All97] Robert J. Allen. A Formal Approach to Software Architecture. PhD Thesis CMU-CS-97-144, School of Computer Science, Carnegie Mellon University, 1997.

[AP11] Alexia Allanic and Emilien Perico. TOPCASED 5.0.0 Collaborative Work Tutorial, 2011.

[AS10] Jenny Abramov and Arnon Sturm. Supporting Layered Architecture Specifications: A Domain Modeling Approach. In Ilia Bider, Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, Lecture Notes in Business Information Processing 50, pages 195–207. Springer Berlin Heidelberg, 2010.

[BA81] J. Dean Brock and William B. Ackerman. Scenarios: A Model of Nondeterminate Computation. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts*, LNCS 107, pages 252–259. Springer Berlin Heidelberg, 1981.

[BBC+07] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, Antonio Domínguez-Brito, Dominic Létourneau, Françis Michaud, and Christian Schlegel. Trends in Component-Based Robotics. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, Springer Tracts in Advanced Robotics 30, chapter 8, pages 135–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BCH+08] Kari Ann Briski, Poonam Chitale, Valerie Hamilton, Allan Pratt, Brian Starr, Jim Veroulis, and Bruce Villard. Minimizing Code Defects to Improve Software Quality and Lower Development Costs. White paper, IBM Rational Software Analyzer and IBM Rational PurifyPlus Software, October 2008.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[BCK+09] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, VietYen Nguyen, Thomas Noll, and Marco Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security*, LNCS 5775, pages 173–186. Springer Berlin Heidelberg, 2009.

[BCK+14] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Panagiotis Katsaros, Konstantinos Mokos, Viet Yen Nguyen, Thomas Noll, Bart Postma, and Marco Roveri. Spacecraft early design validation using formal methods. *Reliability Engineering & System Safety*, 132(0):20 – 35, 2014.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software, Practice, and Experiance*, 36:1257–1284, 2006.

[BCR06] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a Sys-

tem Model for UML. The Structural Data Model. Technical Report TUM-I0612, TU Munich, Germany, June 2006.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDD+93] 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, Technische Univerität München, 1993.

[BEH+02] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML Progress Report Structural Layer Proposal. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, LNCS 2265, pages 501–512. Springer Berlin Heidelberg, 2002.

[BEJV93] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 1993.

[BFLvH08] Christopher Brooks, Thomas Huining Feng, Edward A. Lee, and Reinhard von Hanxleden. Multimodeling: A Preliminary Case Study. Technical Report UCB/EECS-2008-7, Electrical Engineering and Computer Sciences University of California at Berkeley, 2008.

[BGL09] Stefan Björnander, Lars Grunske, and Kristina Lundqvist. Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems*, LNCS 5581, pages 101–115. Springer Berlin Heidelberg, 2009.

[BHH+06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. *Electronic Notes in Theoretical Computer Science*, 160(0):75–96, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus– Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools*. Addison-Wesley, 2000.

[BLF14] Omar Bahy Badreddin, Timothy C. Lethbridge, and Andrew Forward. A Novel

Approach to Versioning and Merging Model and Code Uniformly. In *MODELSWARD*, pages 254–263, 2014.

[Blo08] Joshua Bloch. *Effective Java*. Java Series. Pearson Education, 2nd edition, 2008.

[BMR12] Jan Olaf Blech, Dongyue Mou, and Daniel Ratiu. Reusing Test-Cases on Different Levels of Abstraction in a Model Based Development Tool. In *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012.*, pages 13–27, 2012.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, 1988.

[BOF+10] Kyungmin Bae, Peter Csaba Olveczky, Thomas Huining Feng, Edward A. Lee, and Stavros Tripakis. Verifying Hierarchical Ptolemy II Discrete-Event Models Using Real-Time Maude. Technical Report UCB/EECS-2010-50, Electrical Engineering and Computer Sciences University of California at Berkeley, 2010.

[BR05] Manfred Broy and Andreas Rausch. Das neue V-Modell®XT. *Informatik-Spektrum*, 28(3):220–229, 2005.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[CCO+05] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent Software Verification with States, Events, and Deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[CEP+06] Vaclav Cechticky, Martin Egli, Alessandro Pasetti, O. Rohlik, and Tullio Vardanega. A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications. In Maurizio Morisio, editor, *Reuse of Off-the-Shelf Components*, LNCS 4039, pages 312–325. Springer Berlin Heidelberg, 2006.

[CGR08a] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR08b] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Statecharts. Informatik-Bericht 2008-04, TU Braunschweig, Germany, 2008.

[CHS10] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[Con10] FlexRay Consortium. FlexRay Communications System Electrical Physical Layer Specification Version 3.0.1, 2010.

[Cra96] Donald C. Craig. *Extensible Hierarchical Object-Oriented Logic Simulation*

with an Adaptable Graphical User Interface. PhD thesis, Memorial University of Newfoundland, 1996.

[CRBS09] M. Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In Michel R.V. Chaudron, editor, *Models in Software Engineering*, LNCS 5421, pages 5–19. Springer Berlin Heidelberg, 2009.

[Cza04] Krzysztof Czarnecki. Overview of Generative Software Development. In *Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France*, number 3566 in LNCS, pages 326–341. Springer, 2004.

[DIHK+01] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous Concurrent Modeling and Design in Java. Technical Report UCB/ERL M01/12, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2001.

[DvdHT01] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Working IEEEAFIP Conference on Software Architecture*, pages 103–112, Amsterdam, The Netherlands, August 2001. IEEE.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 266–276, Orlando, Florida, May 2002. ACM.

[DvdHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A Comprehensive Approach for the Development of XML-Based Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005.

[EJL+03] Johan Eker, Jörn W. Janneck, A. Edward Lee, Jie Liu, Xiaojun Liu, Jozef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity – the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Computer-Aided Verification, (CAV '98)*, LNCS 1427, pages 516–520. Springer-Verlag, 1998.

[Fei14] Peter Feiler. An Incremental Life-cycle Assurance Strategy for Critical System Certification. In *Proceedings and Presentations of the Ninth Annual TSP Symposium*. TSP Symposium, November 2014.

[FFL05] Benoît Fraikin, Marc Frappier, and Régine Laleau. State-based versus Event-based Specifications for Information Systems: a Comparison of B and EB^3. *Software and System Modeling*, 4(3):236–257, 2005.

[FG12] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[FGH06] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report Technical Note CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 2006.

[FHKS09] Jan Friedrich, Ulrike Hammerschall, Marco Kuhrmann, and Marc Sihling. Das V-Modell XT. In *Das V-Modell®XT*, Informatik im Fokus, Seiten 1–32. Springer Berlin Heidelberg, 2009.

[FK05] William B. Frakes and Kyo Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.

[FM07] Christoph Ficek and Fabian May. Umsetzung der Java 5 Grammatik für MontiCore. Studienarbeit, Institut für Software Systems Engineering, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2007.

[For82] Internet Engineering Task Force. RFC 826: An Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardwar, November 1982. Updated by RFCs 5227, 5494.

[For89] Internet Engineering Task Force. RFC 1122: Requirements for Internet Hosts – Communication Layers, October 1989.

[For00] Roger Forster. Manchester Encoding: Opposing Definitions Resolved. *Engineering Science and Education Journal*, 9(6):278–280, Dec 2000.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[FS11] Kevin R. Fall and W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Professional Computing Series. Pearson Education, 2011.

[FSD03] Marc Frappier and Richard St-Denis. EB^3: an Entity-based Black-box Specification Method for Information Systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.

[Fuc95] Max Fuchs. Formal Design of a Modulo-N Counter. Technical Report TUM-I9512, Technische Univerität München, 1995.

[Gal04] Daniel Galin. *Software Quality Assurance: From Theory to Implementation*. Pearson education, 2004.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT Softw. Eng. Notes*, 19(5):175–188, December 1994.

[Gar95] David Garlan. An introduction to the Aesop system, 1995.

[GGR06] Boris Gajanovic, Hans Grönniger, and Bernhard Rumpe. *From MDD Concepts to Experiments and Illustrations*, chapter Model Driven Testing of Time Sensitive Distributed Systems, pages 131–148. ISTE Ltd., 2006.

[GH10] Olivier Gilles and Jérôme Hugues. Expressing and Enforcing User-Defined Constraints of AADL Models. In *15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 337–342, March 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, Germany, October 2007.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray, and Joaquim Filipe Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 74–85, Angers, Loire Valley, France, February 2015. SciTePress.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, TU Braunschweig, Deutschland, August 2006.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: a Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97. IBM Press, 1997.

[Hau06] Matthew Hause. The SysML Modelling Language. In *Fifth European Systems Engineering Conference*, September 2006.

[Hei12] Matthias Heinz. *Modellbasierte Entwicklung und Konfiguration des zeitgesteuerten FlexRay Bussystems*. Steinbuch Series on Advances in Information Technology / Karlsruher Institut für Technologie, Institut für Technik der Infor-

mationsverarbeitung. KIT Scientific Publishing, Karlsruhe, 2012.

[Her14] Christoph Herrmann. *Integrierte Software Engineering Services zur effizienten Unterstützung von Entwicklungsprojekten*. Aachener Informatik-Berichte, Software Engineering Band 16. Shaker Verlag, Aachen, Deutschland, 2014.

[HF10] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, LNCS 6100, pages 317–322. Springer Berlin Heidelberg, 2010.

[HG13] Jérôme Hugues and Serban Gheoghe. The AADL Constraint Annex. In *SAE 2013 AeroTech Congress & Exhibition*, Montreal, Canada, September 2013. unpublished.

[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'13)*, pages 22–31, Tokyo, Japan, September 2013. ACM.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *International Journal on Software Tools for Technology Transfer*, pages 1–26, 2015.

[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 (VaMoS'13)*, pages 11–18, Pisa, Italy, January 2013. ACM, New York, NY, USA.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *ECSA '11 5th European Conference on Software Architecture: Companion Volume*, New York, NY, USA, September 2011. ACM New York.

[HLMSN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray, and Joaquim Filipe Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 19–31, Angers, Loire Valley, France, February 2015. SciTePress.

[HLP+14] Florian Hölzl, Christian Leuxner, Birgit Penzenstadler, Martin Haldenmair, Christoph Döbber, and Andreas Wandinger. AutoFOCUS 3 The Picture Book. Technical report, Institut für Informatik, Software and Systems Engineering, TU Munich, 2014. unpublished, 13.12.14.

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hom12] Christoph Hommelsheim. Entwicklung eines erweiterbaren Generator-Frameworks für Modelldokumentationen. Bachelor thesis, RWTH Aachen University, February 2012.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Informatik-Bericht 2010-01, pages 13 – 22, Dagstuhl Castle, Germany, February 2010. fortiss GmbH, Germany.

[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'11)*, pages 150–159, Munich, Germany, August 2011. IEEE Computer Society.

[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 University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10, Munich, Germany, February 2011. fortiss GmbH.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In D. Garlan and R. Calinescu, editors, *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208, Oxford, UK, March 2012. Springer, Germany.

[HST10] Florian Hölzl, Maria Spichkova, and David Trachtenherz. AutoFocus Tool Chain. Technical Report TUM-I1021, Technische Universität München, november 2010.

[HWF+10] Jörgen Hansson, Lutz Wrage, Peter H. Feiler, John Morley, Bruce Lewis, and J'erôme Hugues. Architectural Modeling to Verify Security and Nonfunctional Behavior. *Security Privacy, IEEE*, 8(1):43–49, January 2010.

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Oviedo Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[IEE11] IEEE Standard for Standard SystemC Language Reference Manual, September 2011. `http://standards.ieee.org/getieee/1666/download/1666-2011.pdf`.

[IEE12] IEEE 802.3™: Standard for Ethernet, 2012. `http://standards.ieee.`

org/about/get/802/802.3.html.

[Ix12] Tim Ix. Entwicklung einer wiederverwendbaren Komponentenbibliothek zur Überwachung von Online-Systeme. Bachelor thesis, RWTH Aachen University, 2012.

[JSR14] JSR 360: Connected Limited Device Configuration 8., April 2014. https://www.jcp.org/en/jsr/detail?id=360.

[Kau13] Oliver Kautz. Implementation and Comparison of Distributed System Case Studies Using MontiArc. Bachelor thesis, RWTH Aachen University, 2013.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, editors, *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09)*, Techreport B-108, pages 7–13, Orlando, Florida, USA, October 2009. Helsinki School of Economics.

[KM05] Martin Kempa and Zoltán Adám Mann. Model Driven Architecture. *Informatik-Spektrum*, 28(4):298–302, 2005.

[KND+09] Roy Kawahara, Hiroaki Nakamura, Dolev Dotan, Andrei Kirshin, Takashi Sakairi, Shinichi Hirose, Kohichi Ono, and Hiroshi Ishikawa. Verification of Embedded System's Specification Using Collaborative Simulation of SysML and Simulink Models. In *International Conference on Model-Based Systems Engineering, 2009 (MBSE '09).*, pages 21–28, March 2009.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Deutschland, März 2010.

[KRH+05] Petri Kukkala, Jouni Riihimäki, Marko Hännikäinen, Timo D. Hämäläinen, and Klaus Kronlöf. UML 2.0 Profile for Embedded System Design. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 710–715, Washington, DC, USA, 2005. IEEE Computer Society.

[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)*, Technical Report TR-37, pages 150–158, Portland, Oregon, USA, October 2006. Jyväskylä University, Finland.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, Technical Reports TR-38, Montreal, Quebec, Canada, October 2007. Jyväskylä University, Finland.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of

Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, pages 286–300, 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)*, LNBIP 11, pages 297–315, Zurich, Switzerland, July 2008. Springer, Germany.

[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.

[LAPJ10] Marc M. Lankhorst, Henderik Alex P., and Henk Jonkers. The Anatomy of the Archimate Language. *International Journal of Information System Modeling and Design*, 1(1):1–32, 2010.

[Lee10] Edward A. Lee. Disciplined Heterogeneous Modeling. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, LNCS 6395, pages 273–287. Springer Berlin Heidelberg, 2010.

[Lim94] W.C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *Software, IEEE*, 11(5):23–30, September 1994.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21:336–355, 1995.

[LNPR+13] Markus Look, Antonio Navarro Pérez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In B. Combemale, J. De Antoni, and R. B. France, editors, *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, CEUR Workshop Proceedings 1102, Miami, Florida, USA, 2013.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[MBB95] Walcélio L. Melo, Lionel C. Briand, and Victor R. Basili. Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems. Technical Report CS-TR-3395, University of Maryland, Departement of Computer Science, College Park, MD, USA, January 1995.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[MDT07] Nenad Medvidovic, Eric Dashofy, and Richard N. Taylor. Moving Architectural Description from Under the Technology Lamppost. *Information and Software Technology*, 49(1):12–31, 2007.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, October 1996.

[MKM06] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using Resemblance to Support Component Reuse and Evolution. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 49–56, New York, NY, USA, 2006. ACM.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013.

[Mos09] Pieter J. Mosterman. Elements of a Robotics Research Roadmap: A Model-Based Design Perspective, 2009.

[MSMB04] J. Bret Michael, Man-Tak Shing, Michael H. Miklaski, and Joel D. Babbitt. Modeling and Simulation of System-of-Systems Timing Constraints with UML-RT and OMNeT++. In *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping, 2004.*, pages 202–209, June 2004.

[MSN11] Pedram Mir Seyed Nazari. Architektur Alignment von Java Systemen. Diploma thesis, RWTH Aachen University, September 2011.

[MSUW02] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-Driven Architecture. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, LNCS 2426, pages 290–297. Springer Berlin Heidelberg, 2002.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[NBKN14] Viet Yen Nguyen, Benjamin Bittner, Joost-Pieter Katoen, and Thomas Noll. Compositional Analysis Using Component-Oriented Interpolation. In *Proceedings of the Formal Aspects of Component Software (FACS 2014)*, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J.-M. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, CEUR 1118, pages 15–24, Miami, Florida, USA, 2013. CEUR-WS.org.

[NXD+04] Leila Naslavsky, Lihua Xu, Marcio Dias, Hadar Ziv, and Debra J. Richardson. Extending xADL with Statechart Behavioral Specification. In *Proceedings of the Twin Workshops on Architecting Dependable Systems at International Con-*

ference of Software Engineering, pages 22–26, 2004.

[ÖBM10] Peter C. Ölveczky, Artur Boronat, and José Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, LNCS 6117, pages 47–62. Springer Berlin Heidelberg, 2010.

[OL06] Ian Oliver and Vesa Luukkala. On UML's Composite Structure Diagram. In *Proceedings of the 5th Workshop on System Analysis and Modelling*, June 2006.

[OL07a] Martin Ouimet and Kristina Lundqvist. The TASM Toolset: Specification, Simulation, and Formal Verification of Real-Time Systems. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, LNCS 4590, pages 126–130. Springer Berlin Heidelberg, 2007.

[OL07b] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07)*, 2007.

[ÖM07] Peter C. Ölveczky and José Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[OMG11a] Object Management Group. OMG UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1. `http://www.omg.org/spec/MARTE/1.1/`, 2011. Release date June 2011.

[OMG11b] Object Management Group. OMG Unified Modeling Language (UML), version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, 2011. Release date August 2011.

[OMG12] Object Management Group. OMG Systems Modeling Language (SysML), Version 1.3.. `http://www.omg.org/spec/SysML/1.3/`, 2012. Release date June 2012.

[Pan10] Rajesh K. Pandey. Architectural Description Languages (ADLs) vs UML: A Review. *SIGSOFT Software Engineering Notes*, 35(3):1–5, May 2010.

[PBF+07a] Russell S. Peak, Roger M. Burkhart, Sanford A. Friedenthal, Miyako W. Wilson, Manas Bajaj, and Injoong Kim. Simulation-Based Design Using SysML Part 1: A Parametrics Primer. In *INCOSE International Symposium*, pages 1516–1535. Wiley Online Library, 2007.

[PBF+07b] Russell S. Peak, Roger M. Burkhart, Sanford A. Friedenthal, Miyako W. Wilson, Manas Bajaj, and Injoong Kim. Simulation-Based Design Using SysML Part 2: Celebrating Diversity by Example. In *INCOSE International Symposium*, pages 1535–1556. Wiley Online Library, 2007.

[Pet13] Marian Petre. UML in Practice. In *35th International Conference on Software Engineering (ICSE 2013)*, pages 722–731, San Francisco, CA, USA, May 2013.

[Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Rab13] Rajeevan Rabindran. Model-Driven Development and Simulation of FlexRay-Based Systems Using MontiArc. Diploma thesis, RWTH Aachen University, July 2013.

[RCS13] Jean-Paul Rodrigue, Claude Comtois, and Brian Slack. *The Geography of Transport Systems*. Routledge, New York, New York, NY, USA, 3rd edition, 2013.

[Rei12] Konrad Reif. Bussysteme. In *Automobilelektronik*, Seiten 1–34. Vieweg+Teubner Verlag, 2012.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering Band 19. Shaker Verlag, Aachen, Deutschland, 2014.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON*, pages 1–9, August 1970.

[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.

[RR15] Alexander Roth and Bernhard Rumpe. Towards Product Lining Model-Driven Development Code Generators. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray, and Joaquim Filipe Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 539–545, Angers, Loire Valley, France, February 2015. SciTePress.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. 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 Holger Giese, Michaela Huhn, Jan Philipps, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, pages 30–43, 2013.

[RRW13b] 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.

[RRW13c] 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'13)*, pages 10–12, Karlsruhe, Germany, May 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries. In *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, CEUR Workshop Proceedings 1, pages 26 – 35, Valencia, Spain, September 2014.

[RSG+08] Stephan Reichelt, Karsten Schmidt, Frank Gesele, Nils Seidler, and Wolfram Hardt. Nutzung von FlexRay als zeitgesteuertes automobiles Bussystem im AUTOSAR-Umfeld. In Peter Holleczek and Birgit Vogel-Heuser, Editoren, *Mobilität und Echtzeit*, Informatik aktuell, Seiten 79–87. Springer Berlin Heidelberg, 2008.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, Deutschland, 2te Edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, Deutschland, 2te Edition, Juni 2012.

[SAE11] SAE Standard AS5506/2: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2., January 2011. `http://standards.sae.org/as5506/2/`.

[SAE12] SAE Standard AS5506B: Architecture Analysis and Design Language (AADL V2.1)., September 2012. `http://standards.sae.org/as5506b/`.

[SAE14] SAE Standard SAE AS 5506/1: Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex., July 2014. `http://standards.sae.org/wip/as5506/1a/`.

[Sch01] Bradley Schmerl. xAcme: CMU Acme Extensions to xArch, 2001. `https://www.cs.cmu.edu/~acme/pub/xAcme/guide.pdf`.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Deutschland, 2012.

[Sch13] Stefan Schubert. Development and Implementation of the TCP/IP Stack in MontiArc. Bachelor Thesis, RWTH Aachen University, 2013.

[Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In Frank

Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, Lecture Notes in Computer Science 1474, pages 250–260. Springer Berlin Heidelberg, 1998.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGW94] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons New York, 1994.

[SSL11] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In Daisuke Chugo and Sho Yokota, editors, *Introduction to Modern Robotics*. iConcept Press, 2011.

[SSP08] João Costa Seco, Ricardo Silva, and Margarida Piriquito. ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration. *Computer Science and Information Systems*, 05(2), 2008.

[ŞVE03] Ahmet Y. Şekercioğlu, András Varga, and Gregory K. Egan. Parallel Simulation Made Easy With OMNeT+. In *Proceedings of the 15th European Simulation Symposium (ESS2003)*, October 2003.

[SW99] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System – An AutoFocus Case Study. Technical Report TUM-I9906, Technische Univerität München, 1999.

[tBFGM08] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In Stefan Leue and Pedro Merino, editors, *Formal Methods for Industrial Critical Systems*, LNCS 4916, pages 133–148. Springer Berlin Heidelberg, 2008.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[TSS08] Jean-François Tilman, Romain Sezestre, and Amélie Schyn. Simulation of System Architectures with AADL. In *Proceedings of 4th International Congress on Embedded Real-Time Systems*, ERTS '08, Toulouse, France, 2008.

[Var14] András Varga. OMNeT++ User Manual Version 4.6, November 2014.

[VG07] Markus Völter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 233–242, September 2007.

[VGV09] Roberto Varona-Gómez and Eugenio Villar. AADL Simulation and Performance Analysis in SystemC. In *Proceedings of the 14th IEEE International Conference on the Engineering of Complex Computer Systems*, pages 323–328. IEEE Computer Society, June 2009.

[VGV10] Roberto Varona-Gómez and Eugenio Villar. Aads+: Aadl simulation including the behavioral annex. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 379–384. IEEE Computer Society, March 2010.

[VH08] András Varga and Rudolf Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Deutschland, 2011.

[WH05] Eoin Woods and Rich Hilliard. Architecture Description Languages in Practice Session Report. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 243–246, 2005.

[Wil06] Doris Wild. AutoFocus 2 – Das Bilderbuch. Technical Report TUM-I0610, Institut für Informatik, Technische Universität München, 2006.

[WNS+05] Cédric Wilwert, Nicolas Navet, Ye-Qiong Song, Françoise Simonot-Lion, et al. Design of Automotive X-by-Wire Systems. *The Industrial Communication Technology Handbook*, 2005.

[Woo05] Eoin Woods. Architecture Description Languages and Information Systems Architects: Never the Twain Shall Meet?, February 2005. IFIP WG 2.10 Software Architecture Meeting White Paper.

[www11] The Ptolemy Project website. `http://ptolemy.eecs.berkeley.edu/`, 2011. Accessed 06/2011.

[www12] MontiCore website `http://www.monticore.de`, 2012. Accessed 02/2012.

[www13a] Antlr website `http://www.antlr.org`, 2013. Accessed 01/2013.

[www13b] Freemarker website `http://freemarker.org/`, 2013. Accessed 01/2013.

[www13c] JUnit website `http://junit.org/`, 2013. Accessed 03/2013.

[www13d] Matlab/Simulink website `http://de.mathworks.com/`, 2013. Accessed 12/2013.

[www13e] Xtext website `https://eclipse.org/Xtext/`, 2013. Accessed 12/2013.

[www14a] AADL Inspector website `http://www.ellidiss.com/products/aadl-inspector/`, 2014. Accessed 12/2014.

[www14b] AADL website `http://www.aadl.info`, 2014. Accessed 12/2014.

[www14c] ADML website http://www.opengroup.org/architecture/adml/adml_home.htm, 2014. Accessed 12/2014.

[www14d] Alternating Bit Protocol in the Free On-Line Dictionary of Computing http://foldoc.org/Alternating+bit+protocol, 2014. Accessed 03/2014.

[www14e] ArchStudio website http://isr.uci.edu/projects/archstudio/, 2014. Accessed 12/2014.

[www14f] AutoFocus 3 website http://af3.fortiss.org/, 2014. Accessed 12/2014.

[www14g] COMPASS Project website http://compass.informatik.rwth-aachen.de/, 2014. Accessed 12/2014.

[www14h] CPN-AMI website http://move.lip6.fr/software/CPNAMI/, 2014. Accessed 03/2014.

[www14i] fortis website http://www.fortiss.org/, 2014. Accessed 12/2014.

[www14j] Maven build lifecycle website https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html, 2014. Accessed 05/2014.

[www14k] Maven website https://maven.apache.org/, 2014. Accessed 05/2014.

[www14l] Modelica website https://www.modelica.org/, 2014. Accessed 12/2014.

[www14m] Ocarina website http://www.openaadl.org/ocarina.html, 2014. Accessed 12/2014.

[www14n] OMNeT++ website http://omnetpp.org/, 2014. Accessed 12/2014.

[www14o] Osate 2 website https://wiki.sei.cmu.edu/aadl/index.php?title=Osate_2&oldid=4742, 2014. Accessed 12/2014.

[www14p] RAMSES website http://penelope.enst.fr/aadl/wiki/Projects#RAMSES, 2014. Accessed 12/2014.

[www14q] Ruby on Rails website http://www.rubyonrails.org/, 2014. Accessed 03/2014.

[www14r] SCoPE website http://www.teisa.unican.es/gim/en/scope/scope_web/scope_home.php, 2014. Accessed 12/2014.

[www14s] taste website http://taste.tuxfamily.org/, 2014. Accessed 12/2014.

[www14t] The Acme Project website. https://www.cs.cmu.edu/~acme/, 2014. Accessed 12/2014.

[www14u] The AcmeStudio website. https://www.cs.cmu.edu/~./acme/AcmeStudio/index.html, 2014. Accessed 12/2014.

[www14v] TINA website `http://projects.laas.fr/tina//papers.php`, 2014. Accessed 03/2014.

[www14w] Validas AG website `http://www.validas.de/`, 2014. Accessed 12/2014.

[www14x] xADL website. `http://isr.uci.edu/projects/xarchuci/`, 2014. Accessed 18/2014.

[www14y] xArch website. `http://isr.uci.edu/architecture/xarch/`, 2014. Accessed 18/2014.

[www15a] CDO model repositories website `http://www.eclipse.org/cdo/`, 2015. Accessed 12/2015.

[www15b] Papyrus website `http://www.eclipse.org/papyrus/`, 2015. Accessed 03/2015.

[www15c] PolarSys website `http://www.polarsys.org/`, 2015. Accessed 03/2015.

[www15d] VNuSMV website `http://nusmv.fbk.eu/`, 2015. Accessed 12/2015.

[www15e] Z3 website `https://github.com/Z3Prover/z3`, 2015. Accessed 12/2015.

[Xio02] Yuhong Xiong. *An Extensible Type System for Component-Based Design*. PhD thesis, EECS Department, University of California, Berkeley, 2002.

[YHMP09] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a Formal Semantics for the AADL Behavior Annex. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 1166–1171. IEEE, 2009.

List of Figures

Listings

List of Tables

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. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

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] demonstrate 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, which is described in the two books [Rum11] and [Rum12] 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 [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 is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99] and [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] allows the spe-

cification of 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 discussed 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. [MRR14] provides a precise technique to verify consistency of architectural views against a complete architecture in order to increase reusability. 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 be used to 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 in [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 to 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] defines 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 and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping 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)

Products often 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 product commonalities as well as 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 additive, but also can modify the core. A set of commonly 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]. Furthermore, 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 statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines 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 specification 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 fit 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 [KRR14]. Application classes like Cyber-Physical Systems [HHK+14], 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 [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. 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., collecting data from temperature, cars etc. can now easily be 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, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, 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 *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737. TU Munich, Germany, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 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 *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[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 *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 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*, pages 45–60. 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] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 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*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 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. Informatik-Bericht 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 *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[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 (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[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 *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 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 *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[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 Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 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 University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 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. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, 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 *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, 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 *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 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 *Conference on Foundations of Software Engineering (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 *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (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 *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 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, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin, 2te Edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin, 2te Edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 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 *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, 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 *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.