Model-Based Specification of Component Behavior with Controlled Underspecification

Jan O. Ringert[1] and Bernhard Rumpe[2] and Andreas Wortmann[2]

Abstract: The development of distributed interactive systems is a challenging task. Hierarchical decomposition of the systems' functionality into components and explicit identification of their interaction help to reduce this complexity. The behavior of components can be defined using automata and specified and verified per component. However, some properties about the interaction behavior of components in the context of larger systems are not easily specified on a component level and might even crosscut their hierarchical decomposition. We present mechanisms that enable controlled underspecification and allow component models to be used as behavior specifications for complete systems and subsystems crosscutting hierarchical component composition. Behavior specification and implementation in the same modeling language enable a seamless development process based on stepwise refinement following the FOCUS methodology. We have developed a specification language to define crosscutting behavior relations and implemented a prototype for refinement and equality checking using a reduction to the model checker Mona.

1 Introduction

Sophisticated software consists of collaborating distributed systems. Engineering such systems can be facilitated by structuring these as component and connector (C&C) architectures [TMD09, Ma13]. C&C architectures describe the logical decomposition of a system into components, to encapsulate functionality within well-defined interfaces, and connectors, to enable component interaction. We consider a model-driven engineering (MDE) setting where component behavior is defined by composition or modeled using automata. Examples of modeling languages supporting this setting are AutoFOCUS [HF07], Simulink Stateflow [wwwa], UML/SysML [Ob12], and MontiArcAutomaton [RRW14]. MontiArcAutomaton components support a type system with instantiation, yield stable interfaces of sets of typed input and output ports, contain local variables, and are either atomic or hierarchically composed.

The development of complex software components requires multiple steps: Initially, the complete behavior of a component might be unknown and thus underspecified in early development artifacts. During development and evolution information is added and undesired behavior is removed. Encapsulation mechanisms of components allow not only modification of the behavior of a system but also of its logical structure. We adopt the development methodology of FOCUS [BS01] which stipulates development of interactive systems in a stepwise refinement process based on a system model of stream processing components. Streams capture the behavior of components as their observable interaction.

[1] Software Engineering, RWTH Aachen University & School of Computer Science, Tel Aviv University, Israel
[2] Software Engineering, RWTH Aachen University

[RRW16] J. O. Ringert, B. Rumpe, A. Wortmann:
Model-Based Specification of Component Behavior with Controlled Underspecification.
In: Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16), pages 1–12. GI, 2016.
www.se-rwth.de/publications

MDE advocates models as primary development artifacts for requirements specification, testing, model checking, and code generation [Ru12]. We present extensions of MontiArcAutomaton supporting the use of C&C models and automata for specification purposes. Our contribution provides (1) mechanisms for automata that enable controlled underspecification of component behavior and (2) a language for defining refinement and equality relations of components crosscutting the hierarchical system decomposition.

Our mechanisms of controlled underspecification on the one hand allow to leave out implementation details in early models of a software system. On the other hand, even for a concrete implementation, they allows to focus on specific aspects of the system while leaving the rest underspecified. The model-based specification techniques can thus be applied by engineers for top-down refinement from an early specification to an implementation as well as for writing specifications of properties an implementation should satisfy.

The underspecification mechanisms for automata are based on [Ru96] and include omitting states, transitions, port assignments and variable assignments. They are controlled by powerful completions. Our new specification language allows to relate behavior specifications to composed behaviors in the context of a complete software architecture. These crosscutting specifications allow to assert behavioral properties of the current system composition and context that are otherwise not made explicit and might not hold in other compositions of the same components. Our work employs the FOCUS calculus [BR07, BS01] on infinite discrete message streams for synchronous communication.

We provide a prototype for translating MontiArcAutomaton specification and implementation models into predicates for the Mona [EKM98] model checker to enable their analysis.

Sect. 2 gives examples of the analysis enabled by our framework. Sect. 3 presents the MontiArcAutomaton ADL. Sect. 4 describes our approach and Sect. 5 provides information on a prototype implementation. Sect. 6 discusses the prototype and performance results. Sect. 7 highlights related work and Sect. 8 concludes.

2 Specification Modeling Example

We present an example for the development of a software controller for an elevator system based on [SW99]. The elevator system comprises of three floors and the elevator car with a motor and a door. The floors have buttons to request the elevator.

Engineers have developed a C&C software architecture of the system depicted in Fig. 1 and have defined basic responsibilities of the components. The system consists of a composed component type `ECS` with two subcomponents `floors` and `elevator`. Subcomponent `floors` contains three subcomponents of type `Floor` responsible for translating pressed buttons into requests for the subcomponent `elevator` and activating lights on respective floors. Component type `Elevator` comprises of a subcomponent `control` that handles requests via components `motor` and `door` depending on position information read from the environment. Once a floor is reached, `control` should emit a clear request to the corresponding `Floor` component. In this architecture, the incoming port `dist` of component `control` is not

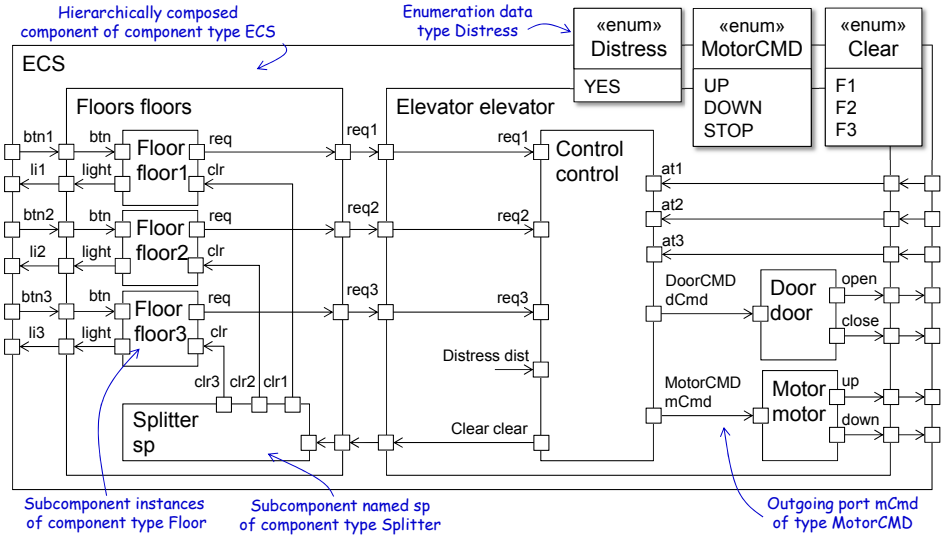

Fig. 1: The elevator system comprising of floors and an elevator with motor and door

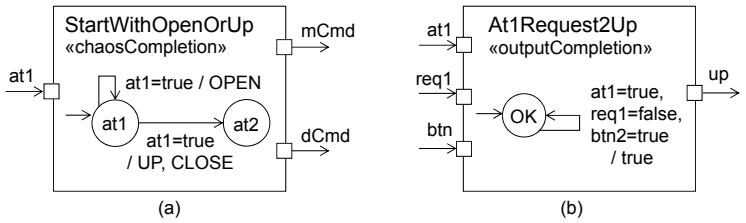

Fig. 2: Two specifications: Specification `StartWithOpenOrUp` (a) describes the initial behavior of component `Control` and hence is connected its ports `at1`, `mCmd`, and `dCmd`. Specification `At1Request2Up` (b) is connected to the ports `at1` and `req1` of component `control`, `btn` of component `floor2`, and `up` of component `motor` to describe crosscutting behavior.

connected. In an extended variant `control` should stop the motors and open the door when a distress message arrived on port `dist`. Ports are of type Boolean if not denoted otherwise (e.g., port `mCmd` of component `Control` is of data type `MotorCMD`).

One requirement for the behavior of the elevator is that if it is initially on the first floor it should either open the door or close it and move up. After moving up no behavior is specified, yet. This requirement is formalized as the automaton shown in component `StartWithOpenOrUp` Fig. 2 (a). In case the input at port `at1` is `true` in the initial state two alternative transitions may be executed. If no transition is enabled, i.e., no behavior is specified, the future behavior is unrestricted (denoted by `«chaosCompletion»`). The required behavior only restricts component `Control`, the engineers thus assert that `Control` refines `StartWithOpenOrUp` on common ports.

Another requirement is, that whenever the elevator resides at the first floor and there is no active request for this floor, but one for the second floor, then the elevator will start to drive upward. An engineer expresses the required behavior in form of the component

At1Request2Up shown in Fig. 2 (b), which specifies behavior crosscutting the C&C model. Messages arriving on input ports at1, req1 of component control and the input port btn of component floor2 determine the output on port up of component motor. The stereotype «outputCompletion» in Fig. 2 (b) states that the component allows additional transitions whenever the behavior of the component is not defined by an existing transition, e.g., when the elevator car is not in the first floor.

Note the difference between the first refinement example for component Controller and the second example crosscutting many components of the C&C model. For the specification At1Request2Up it is important to check refinement in the context of the overall system. For example, the check would fail if Distress messages stop upward movements.

3 Components and Automata in MontiArcAutomaton

MontiArcAutomaton is a C&C ADL developed with the language workbench MontiCore [KRV10] for modeling hierarchically composed components with unidirectional connectors between typed input and output ports of components. Components have a set of input and output ports and are either composed or atomic. Composed components have subcomponents and connectors. Atomic components have a set of local variables and an automaton describing component behavior. Ports and variables have a name and a type. For detailed definitions and well-formedness rules see [HRR12, RRW14, Ri14].

Atomic components embed automata to describe component behavior. An automaton comprises a finite set of states, initial variable values, a set of initial states with optional outputs, and a set of transitions. Every transition has a source state, a pattern of values read on input ports (inputs) and local variables, a target state, values written to output ports (outputs), and values assigned to local variables (assignments). Inputs, outputs, and assignments may refer to values read on input ports and values of variables.

A semantic domain for MAA automata are predicates over infinite discrete streams following the FOCUS paradigm [BS01, RR11]. Component semantics are predicates over one input stream for every input port of the component and one output stream for every output port.

All components are executed synchronously to the tick of a global clock and the position $t \in \mathbb{N}$ in a stream contains the message transmitted at time point t. We distinguish weak causality, where a component can instantaneously react to an input received at time t, from strong causality, where a reaction happens at $t+1$ or later [BS01] (also called strict causality). Frameworks allowing weakly causal behavior typically require engineers to add at least one strongly causal component to every feedback cycle. We give semantics of MAA automata in Sect. 5 that lead to strongly causal behavior if the component specifies initial outputs and weakly causal behavior otherwise.

The first elements on the output streams are, if provided, the initial outputs of the automaton. Inductively, the current state and inputs at positions $t \in \mathbb{N}_\infty$ of the input streams and the values of local variables determine the enabled transitions of the automaton. The

execution of a transition determines the state, variable assignments, and values of the output streams at positions $t+1$ for strong causality and t for weak causality. References to ports and variables are interpreted as their values at time t. Nondeterminism is induced by free choice among enabled transitions and expressed in I/O relation semantics as multiple possible output streams (complete definitions available from [Ri14]).

4 Specification Driven Architecture Modeling

We use automata and composition for the specification and implementation of behavior. Underspecification of component behavior is supported by various mechanisms: nondeterminism of transitions, incomplete patterns for enabling transitions, incomplete definition of outputs and variable assignments, and several variants for a completion of the transition system in cases where no transition is enabled. The specification for a component may refer to a subset of input and output ports of the implementation as in the example in Sect. 2 for component `StartWithOpenOrUp` restricting only messages on some ports of component `Controller`.

These model-based specification techniques can be applied by engineers for top-down refinement from an early specification to an implementation as well as for writing specifications of properties an implementation should satisfy.

Transition System Completions to Control Underspecification

Automata inside MontiArcAutomaton components may describes a superset of the behaviors allowed by a component or subsystem implementation. Except for the source state all elements of transitions are optional. This possible incompleteness is interpreted as underspecification. MontiArcAutomaton uses stereotypes to control underspecified behavior by completions of the transition systems of automata. The specification framework supports the stereotypes «`nilCompletion`», «`chaosCompletion`», and «`outputCompletion`». We describe the intuition of completions. Definitions are available from [Ri14].

The default completion for automata used as implementations is «`nilCompletion`». Undefined behavior (no enabled transition) preserves the state, preserves variable values, and sends empty messages.

Chaos completion of MontiArcAutomaton automata allows arbitrary outputs on ports with previously omitted outputs for initial states and existing transitions. In addition, a new state *Chaos* is added and the transition relation is completed with arbitrary transitions for cases where no existing transitions was enabled. Choosing arbitrary transitions produces a chaotic component state, hence we denote it as chaos completion. Chaos completion is applied to the automaton shown in Fig. 2 (a).

Output completion is a refinement of chaos completion that allows arbitrary outputs in case no behavior is defined by the original automaton. In contrast to chaos completion the automaton remains in its current state and preserves all variable values. It thus resumes in the most recent state when it encounters input patterns for which its reaction is defined.

Output completion is applied to component `At1Request2Up` (Fig. 2 (b)). Applying chaos completion instead would have restricted behavior only in the initial state.

Component Composition and Exhibiting Internal Channels

Semantics of atomic and composed MontiArcAutomaton components are defined as predicates over input and output streams following [BS01]. Component composition is simply defined as the conjunction of the subcomponent predicates over common channels. Composition yields again a predicate over input and output streams. For a uniform handling of atomic and composed MontiArcAutomaton components we denote the semantics of a component c by $[\![c]\!]$. The predicate semantics of the composed component `ECS` shown in Fig. 1 is defined via the semantics of its subcomponents `Floors` and `Elevator` as:

$$
\begin{aligned}
&[\![\texttt{ECS}]\!](btn1,\ldots,at1,\ldots,li1,\ldots,open,close,up,down) \Leftrightarrow \\
&\qquad \exists req1 \in \mathbb{B}^\infty, req2 \in \mathbb{B}^\infty, req3 \in \mathbb{B}^\infty, clear \in \text{Clear}^\infty : \\
&\qquad\quad [\![\texttt{Floors}]\!](btn1,\ldots,clear,req1,\ldots,li1,\ldots) \wedge \\
&\qquad\qquad [\![\texttt{Elevator}]\!](req1,\ldots,at1,\ldots,clear,open,close,up,down) \qquad (1)
\end{aligned}
$$

For a component c and a set of ports P we define $[\![c]\!]_P$ as component semantics that in addition exhibits the streams on ports P inside the component composition. We identify ports by their qualified name starting from the parent component. This notation allows to expose the messages sent or received by all nested (sub-)components. In the example above the predicate $[\![ECS]\!]_{\{elevator.req1\}}$ exposes the stream sent on port `req1` of subcomponent `elevator` of component `ECS`.

Refinement on Common Channels

Refinement between two specifications states that the refining specification obeys all properties that the refined specification has promised, but it may be more deterministic and more detailed in certain behavior. Subsequent refinement finally leads to a sufficiently detailed implementation. We use components as specifications. Formally, a component c' refines a component c if all I/O histories i,o that c' allows are also allowed by c, schematically, $\forall i,o : [\![c']\!](i,o) \Rightarrow [\![c]\!](i,o)$.

Our framework supports classical behavior refinement and also interface refinement where the implementation might have a reduced input interface and an extended output interface and can thus replace the component it refines [Br93]. For specifications defining partial behavior, we also support the case where a specification constrains a subset of the input and output streams of a system. This is a special case of upward simulation [Br93]. In addition to refinement we also consider also equality. Equality requires that all I/O histories in the semantics of one component are exactly those in the semantics of another.

An example of refinement on a subset of input and output streams has been presented in Sect. 2 for `ECS` and `At1Request2Up` shown in Fig. 1 and Fig. 2 (b). We state the refinement given in the example below. It requires component `ECS` to exhibit the internal

stream on port ECS.elevator.req1 as universally quantified variable $req1'$.

$$\forall btn1, .., down, req1' : \\ [\![ECS]\!]_{\{elevator.req1\}}(btn1, .., req1') \Rightarrow [\![\texttt{At1Request2Up}]\!](at1, req1', btn2, up) \quad (2)$$

MAA Specification Checks

To describe intended relations and their checks we introduce specification checks. A check can optionally be negated and express equality or refinement. A mapping of port names relates the I/O histories of (sub-)components on the left side of the check to ports of components with possibly different interfaces. The refinement example `At1Request2Up` in Sect. 4 is expressed in the next line and followed by a port mapping:

```
ECS refines At1Request2Up where
  elevator.control.req1 -> req1, btn2 -> btn;
```

MAA specification checks define a mapping of ports between two behavior models (typically implementation and specification). This mapping allows the specifier to focus on specific parts of the system and analyze them in the context of the overall composition. A default mapping is based on same named ports of compared components, e.g., ports `up` and `at1` in the above example. Some interesting cases occur when not all ports on the left and right are mapped. In our example (Sect. 2) the specification constrains a subset of the inputs and outputs of the system. Here, for all input received on unmapped ports the output of a component has to be allowed by its specification. The output produced on unmapped ports is not relevant for satisfaction of the check. Checking refinement for substituting a component c' by c sometimes allows c to ignore inputs and produce additional outputs [Br93]. In a corresponding MAA specification check c `refines` c' inputs of c' and outputs of c might be unmapped. For all additional inputs c' might have, c has to produce output allowed by c'.

5 Specification Checking using Mona

We have implemented a prototype to check MontiArcAutomaton specifications based on a translation of components into weak monadic second order logic with one successor (WS1S) implemented in the model checker Mona [EKM98]. Elements of WS1S formulas are natural numbers (first order) with the successor function and finite sets of natural numbers (second order) with usual set operations. The syntax of Mona allows the definition of predicates and the inclusion of declarations from external files. Mona can compute (minimal) assignments of free variables in the formula for its satisfaction or non-satisfaction. A key idea of our solution is the representation of FOCUS semantics of components, i.e., relations of input and output streams, as predicates over sets of natural numbers. This enables a compositional translation of components, declarative support for chaos, output, and nil completion of transition systems, and checking of specifications.

Mona

```
1 var2 clear_F1, clear_F2, clear_F3;
2 assert clear_F1 inter (clear_F2 union clear_F3) = empty;
3 assert clear_F3 inter clear_F3 = empty;
```

Listing 1: The stream *clear* $\in$ Clear* encoded in Mona

Mona

```
1  pred At1Request2Up(var2 req1_false,var2 req1_true,
2    var2 btn_false,var2 btn_true, var2 at1_false,var2 at1_true,
3    var2 up_false, var2 up_true, var2 allTime) =
4  ex2 OK: #state of automaton
5    allTime sub OK & # defined state at least for all points in time
6    (0 in OK) & #initial state/output
7    all1 t: t in allTime =>
8      ( t in OK & t in at1_true & t in req1_false & t in btn_true
9        & t in up_true & t+1 in OK)
10     #unrestricted output if no transition is enabled
11     | ( ~(t in OK & t in at1_true & t in req1_false & t in btn_true)
12        & sameNextValue(OK, t) ); # stay in same state
```

Listing 2: Translation of At1Request2Up shown in Fig. 2 (b) (ll. 11-12 shortened)

Streams in Mona

We adapt an encoding of message streams in Mona from [Sc09]. The type of a streams may only have finitely many values. For every finite stream $s \in T^*$ and each value $v \in T$ we declare a set variable in Mona, i.e., a set of natural numbers, which contains a number $t \in \mathbb{N}$ iff the stream s has the value v at time t. An example for the encoding of the stream *clear* $\in$ Clear* to specify on which floor to clear the light (values F1, F2, F3) is shown in Lst. 1. One set (var2) per value is defined in l. 1. The assertions in ll. 2-3 require that the stream has never more than one value at any point in time. The absence of $t \in \mathbb{N}$ from all sets is interpreted as an empty message.

Please note that all sets in WS1S are finite but unbounded. We can thus only represent finite streams T^* instead of infinite streams T^∞. The time-synchronous model of streams used in our semantics, however, requires infinite streams. For proving refinement and equality for infinite streams in Mona we adopt a well-known result form model checking that finite trace containment of finite transition systems implies also containment for infinite traces (see, e.g., [BK08, Ch. 3]) to lift analysis results to infinite computations.

Components, Automata, and Completions in Mona

All MontiArcAutomaton components are translated to Mona predicates with parameters for their input and output streams. Translation of the atomic component At1Request2Up depicted in Fig. 2 (b) is shown in Lst. 2. The head of the predicate in ll. 1-3 defines parameters for the input streams on ports at1, req1, and btn and the output stream on the port up. The parameter allTime is used for synchronization of all components on a global time. The body of the predicate (ll. 4-12) starts with declaring a sequence of states (ll. 4-5). Line 7 restricts the constraints implied by the transitions of the automaton to a synchronized finite time (the times t in allTime).

Constraints implied by the transition of the automaton are defined in ll. 8-9. The constraints relate state and input at time `t` to output at time `t` (weak causality) and state at time `t+1`. Output completion is defined in l. 11 as negation of enabled transitions and preserving the current state. This translation into a declarative logic prevents the explicit expansion of the transition system as defined for the completions.

Composed components are uniformly and systematically translated into predicates with signatures based on their input and output ports. The body of the predicate is the instantiation of the predicates of subcomponents as in Eqn. (1). Importantly, this uniform handling allows compositional and incremental generation of Mona predicates, except for cases of exhibiting streams on internal ports.

The translation of a check into a Mona formula is straightforward: equality or refinement translates into implication for refinement as in Eqn. (2) and equivalence for equality. The mapping of ports defines the parameters for the instantiation of the component predicates on the left and right side of the check.

6 Performance Results on Example Systems and Discussion

We have started evaluation of the performance of the specification checking process, based on systematic checks for six software architectures of three example systems:[1] First example is the elevator system introduced in Sect. 2. It consists of 8 component definitions, with 3 composed components, and 5 components with automata. The second example comprises four architectures of a mobile robot [RRW13], consisting of 15 component definitions with 7 composed components, and 6 atomic components with automata. Third example is the architecture of a pump station taken from AutoFOCUS [HF07] consisting of 14 component definitions with 3 composed components, 10 atomic components with automata, and 1 component with manual implementation.

For each component we defined three specification checks: (ES) checks a component's equality to itself; (RC) checks refinement of a component with arbitrary behavior (chaos); (EC) checks equality with arbitrary behavior. We measured verification times of the generated Mona programs. For the experiments, we used a computer with 2.7 GHz Intel Core i7 CPU, 16GB Ram, Windows 7, and Mona 1.4-13.

Specification checks for the elevator system took 49-70ms (ES), 58-123ms (RC), and 59-120ms (ES) when they completed. The checks RC and ES failed for the components `ECS` and `Floors` due to lack of memory. All checks for the mobile robot succeeded in 55-78ms (ES), 53-157ms (RC), and 58-143ms (EC). Specification checks for the pump station example took 50-123ms (ES). Checking (RC) took 50-1290ms and failed for two specifications. The (EC) checks required from 52-1616ms and failed for two specifications.

This evaluation shows that the Mona implementation handles small examples fast. However, it also shows clear limitations on examples with many streams or large domains. In

[1] All software architectures and specification checks are available from [wwwb].

particular, the analysis does not take longer time but simply fails due to memory limitations. We conclude that the current prototype can not be applied to real examples.

We had chosen Mona for the prototype because of the natural formulation of streams and components in WS1S. These notions coincide with the formal definitions of the FOCUS framework and thus reduce the translation complexity while supporting readability and manual validation. Although the complexity of solving W1S1 problems is non-elementary, the benefits of a fully automated decision procedure and example computation legitimize to use Mona as a research prototype. The results of our evaluation however show limitations of the prototype for practical evaluation and application. It is important to note that the non-elementary complexity of the prototype is not the complexity of the refinement problem which is PSPACE-hard and can thus be solved in exponential time. We consider reformulating MAA specification checks in other formalisms to apply solvers with lower space and time complexities.

7 Related Work

We briefly mention underspecification and refinement mechanisms in state-based formalisms and analyses of modeling languages similar to MontiArcAutomaton. A more thorough discussion of related work can be found in [Ri14].

Many types of automata provide their own mechanisms for composition and refinement. Modal transition systems [La89] define *may* and *must* transitions with a number of refinement definitions [LNW07] for preserving *must*-behavior and allowing to remove *may*-behavior. Refinement for interface theories [Al05] is defined for equivalent signatures (similar to our component interfaces) based on alternating simulation with contravariant input and covariant output in similar states. Most refinement notions for modal transition systems rely on simulation relations [LNW07]. In modal transition systems and interface theories behavior is modeled explicitly where our approach offers completions. We rely on I/O streams containment as refinement following the FOCUS theory because it natively combines automata and component composition with theories of interface and behavior refinement [BS01].

Combinations of C&C modeling and automata are available in many languages, e.g., SysML's [Ob12] internal block diagrams, in the AutoFOCUS IDE [HF07] and in Simulink Stateflow [wwwa]. Formal analysis approaches for these languages exist, mainly based on model checking [HSE97, Sc09, LMÁ09, ERB13] for consistency against temporal logic constraints. We are not aware of approaches supporting crosscutting specifications and analysis as in MAA specification checks.

8 Conclusion

We presented mechanisms for controlled underspecification of behavior provided by MontiArcAutomaton and its embedded automata. These mechanisms fully support component

instantiation and composition and enable the specification and stepwise refinement of component behavior. Importantly, specifications may crosscut the component hierarchy to define properties of components in the restricted setting of their current composition. A prototype implementation for defining and evaluating specification checks provides a seamless model-based integration in the MontiArcAutomaton framework from specification to implementation. The current prototype suggests future work on improving scalability.

References

[Al05] de Alfaro, Luca; da Silva, Leandro Dias; Faella, Marco; Legay, Axel; Roy, Pritam; Sorea, Maria: Sociable Interfaces. In: Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005. 2005.

[BK08] Baier, Christel; Katoen, Joost-Pieter: Principles of Model Checking. The MIT Press, 2008.

[Br93] Broy, Manfred: (Inter-)Action Refinement: The Easy Way. In: Program Design Calculi. 1993.

[BR07] Broy, Manfred; Rumpe, Bernhard: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik Spektrum, 2007.

[BS01] Broy, Manfred; Stølen, Ketil: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg, 2001.

[EKM98] Elgaard, Jacob; Klarlund, Nils; Møller, Anders: MONA 1.x: new techniques for WS1S and WS2S. In: Computer-Aided Verification, (CAV '98). 1998.

[ERB13] Elberzhager, Frank; Rosbach, Alla; Bauer, Thomas: Analysis and Testing of Matlab Simulink Models: A Systematic Mapping Study. In: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation. 2013.

[HF07] Hölzl, Florian; Feilkas, Martin: AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In: Model-Based Engineering of Embedded Real-Time Systems. 2007.

[HRR12] Haber, Arne; Ringert, Jan Oliver; Rumpe, Bernard: MontiArc – Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical report, RWTH Aachen, 2012.

[HSE97] Huber, Franz; Schätz, Bernhard; Einert, Geralf: Consistent graphical specification of distributed systems. FME'97: Industrial Applications and Strengthened Foundations of Formal Methods, 1997.

[KRV10] Krahn, Holger; Rumpe, Bernhard; Völkel, Steven: MontiCore: a framework for compositional development of domain specific languages. STTT, 2010.

[La89] Larsen, Kim Guldstrand: Modal Specifications. In: Automatic Verification Methods for Finite State Systems. 1989.

[LMÁ09] Lucas, Francisco J.; Molina, Fernando; Álvarez, José Ambrosio Toval: A systematic review of UML model consistency management. Information & Software Technology, 2009.

[LNW07] Larsen, Kim Guldstrand; Nyman, Ulrik; Wasowski, Andrzej: On Modal Refinement and Consistency. In: CONCUR 2007, Lisbon, Portugal, Sept. 3-8, 2007. 2007.

[Ma13] Malavolta, Ivano; Lago, Patricia; Muccini, Henry; Pelliccione, Patrizio; Tang, Antony: What Industry Needs from Architectural Languages: A Survey. IEEE Trans. Software Eng., 39(6):869–891, 2013.

[Ob12] Object Management Group (OMG): , OMG Systems Modeling Language (OMG SysML). http://www.omg.org/spec/SysML/1.3/, 2012. Accessed 3/2014.

[Ri14] Ringert, Jan Oliver: Analysis and Synthesis of Interactive Component and Connector Systems. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RR11] Ringert, Jan Oliver; Rumpe, Bernhard: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics, 2011.

[RRW13] Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In: Software Engineering 2013 Workshopband. 2013.

[RRW14] Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. Aachener Informatik-Berichte, Software Engineering 20. Shaker Verlag, 2014.

[Ru96] Rumpe, Bernhard: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Verlag Wissenschaft, 1996.

[Ru12] Rumpe, Bernhard: Agile Modellierung mit UML. Springer, 2 edition, 2012.

[Sc09] Schätz, Bernhard: Model-Based Development of Software Systems: From Models to Tools. Technische Universität München, 2009. Habilitation Thesis.

[SW99] Strobl, Frank; Wisspeintner, Alexander: Specification of an Elevator Control System. Technical report, Technische Universität München, 1999.

[TMD09] Taylor, Richard N.; Medvidovic, Nenad; Dashofy, Eric: Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.

[wwwa] MathWorks Simulink and Stateflow. http://www.mathworks.com. Accessed 3/2014.

[wwwb] MontiArcAutomaton Verification website. http://www.monticore.de/languages/montiarcautomaton/verification/. Contains supporting materials for this paper.