Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Model-driven design and verification of component-based reactive systems

Scientific Students' Association Report

Author:

Bence Graics

Advisor:

Molnár Vince
Vörös András

2016

Contents

Abstract

We are surrounded by a large number of safety critical systems such as railway, cars and aircrafts. The incorrect behavior of such systems may have serious consequences, even to the extent of threatening human lives, so we need techniques supporting the design and development of correct systems. The application of *model-driven paradigms* is getting more and more important as the complexity of such systems have increased rapidly which could not be managed by traditional development methods. The main advantage of model-driven approaches is that not only do they document the components of the system, but implementation can be derived automatically using *code generation*. Several tools and languages are available supporting the design of systems with models. The internal behavior of reactive systems are usually represented by state-based models, starting from the component-level and using composition to build the system-level model. Unfortunately, many of the tools that support composition fail to define the precise semantics, making automatic code generation infeasible. Precise validation and formal verification of the design models are rarely supported for the same reason.

Proving correctness is an important requirement when designing safety critical systems. In addition to testing, *formal methods* can be applied to verify the correctness of the system design in an early phase. A common approach to state-based behavior analysis is *model checking*. Unfortunately, most of the modeling formalisms tailored for engineers are not suitable for direct analysis, therefore formal models usually have to be created manually by an expert team.

The goal of this work is to develop a framework that supports the design and analysis of *state-based behavioral* models. Based on an intermediate statechart language, a new language is defined to facilitate the *composition of statechart models* with precise semantics. The framework includes a code generator that produces the implementation of the composed system, assuming the implementation of the statechart models are given (as most tools support code generation for a single statechart) and following the semantics of the compositional language. To support the modeling process, validation rules have been defined for the intermediate statechart language to find design flaws as soon as possible. Furthermore, the automatic transformation of individual statecharts as well as their composition to formal models has been developed to support the formal analysis of the design models.

The framework currently builds on Yakindu, an open-source state-based modeling tool. Transformation from Yakindu statechart models to intermediate formal models, as well as from intermediate formal models to UPPAAL formal automata is implemented by model transformations. The validation rules have been developed by using graph pattern matching languages and algorithms. One of the main advantages of the framework is that it is extensible with arbitrary state-based engineering and formal modeling languages, so it can be integrated with other design and analysis tools. The application and the merits

of the framework are demonstrated in a project of the Fault Tolerant Research Group which includes the design and analysis of a distributed railway interlocking system.

Chapter 1

Introduction

As the complexity of software systems increases, more and more responsibility falls onto system engineers who have to supervise the design, implementation and analysis of interacting system components. Development paradigms, such as *model-driven development* have been adopted in order to ease the work of system designers as well as making the development process more transparent. As a motivation of the presented framework, this chapter introduces the model-driven development paradigm, the challenges of system design and offers a possible solution to help designers overcome them.

1.1 Model-driven software development

Modeling is a popular tool in several fields of study, therefore it has many definitions. In this paper work, we use the term *model* in the following sense:

> A model is the simplified image of an element of the real or a hypothetical world (the system), that replaces the the system in certain considerations.

Model-driven software development is the controlled and formalized application of modeling to affect and support system requirements, design, analysis, verification and validation activities [14]. It begins in the conceptual design phase and continues throughout the development and later life cycle phases. The primary artifact of this paradigm is the *model*, which is the main information source in each phase of software development. [10]

Model-driven software development is supposed to replace the document-centric paradigms that have been practiced by systems engineers in the last few decades. Furthermore, it is expected to influence the future routines of system engineering by being entirely integrated into the definition of software development processes. [8]

In model-based approaches the focus is transferred from detailed documentation to the creation and handling of coherent system models. From these models platform-specific source code and documentation can be generated mostly automatically, thus the complexity of the system becomes manageable. Also, the correct behavior of the system can be ensured early in the design phase with the use of verification techniques.

One of the biggest advantages of model-driven software design is code generation. This means that *source code* implementing the behavior of platform-independent models can automatically be generated using code generators. This way the consistency between source code and system models is given and is easily maintainable, as the change of models

automatically propagates to change the underlying code. Furthermore, it minimizes the number of human errors that is typical during the implementation phase.

Model-based approaches are usually applied in the development of safety-critical system, since the validation and verification of the system is mandatory at each phase of the design. This need has led to special development processes: the widespread V-model and the more specialized Y-model that relies on the automatic generation of artifacts.

1.2 Challenges of system design

As softwares are becoming more and more complex, their development gets more circumstantial. In order to support system engineers in the construction of complex systems, Architectural description languages (ADLs) can be used. ADLs tend to focus either on implementation or analysis, but rarely on both of them at the same time. Focusing only on implementation raises difficulties when verification of interactive components is desired. On the other hand, considering only analysis aspects might hinder the implementation of the system.

Even if some ADLs have similar focus, they usually define unique semantics for the composition, or do not define semantics explicitly at all. This prevents the integration of multiple ADLs in the same project, leaving system designers with the trade-off offered by different tools. Tracing and in particular traceability also suffers if multiple tools are used. Typically, one ADL is used throughout the system design process and thus the chosen one might have a large effect on the following development phases.

Modeling the emergent behavior of diverse components can be a particularly important feature in an ADL tool. Model-driven design enables the analysis of emergent behavior of interacting system components. It can reveal unexpected behavior, even if the components have been verified and proven well-behaving on their own. This facilitates the correction of the system at an early stage of software design, sparing valuable resources like budget and time. In spite of the advantages, tools rarely provide such features.

1.3 Overview

As we have seen in Section 1.2 the proliferation of architecture languages, their different or not explicitly defined semantics, their complexity and lack of support for implementation or analysis might be confusing or might not cater to all needs. We have identified the following requirements.

- There is a need for multiple languages as most of them have different advantages and disadvantages. Moreover, it should be possible to compose models built in different formalisms, i.e. to build heterogeneous system models.
- There should be a way to convert between different formalisms in order to leverage the benefits of different tools. Proper conversion can be achieved only if the semantics of every model element is precisely defined. This would sometimes mean the restriction of the set of transformable elements.

The well-defined semantics would enable source code generation, as the exact meaning of elements of the restricted subset of languages would be defined. In addition, a common

semantics would also enable the analysis of the composite behavior of components at the same time, letting the designer solve each problem with the proper tools.

This work presents our solution satisfying the above requirements. The described framework would suit the V-model that is well-known in safety-critical development community.

The V-model defines three design phases. The first phase is *requirement analysis*, where requirements of the system are identified and collected, while the design of acceptance tests is carried out simultaneously. The second phase is *system design*, where the system architecture is defined and the integration tests are prepared. The third phase is *component design* where the low-level internal behavior of individual components is designed and analyzed, and unit tests are often generated. At this point the system design is completed and the system is implemented and deployed based on the produced models and documentation. On the right side of the "V", the implementation is tested by the artifacts produced during the design phases to ensure that the design was properly followed (*verification*) and the proper product was built (*validation*).

The framework presented in this work aims to assist the phases of the V-model by combining the models produced during system design and component design to achieve the following:

- Provide validation during the component design phase to prevent common modeling flaws;
- Automatically generate source code to facilitate implementation and deployment of the system;
- Verify and validate individual component behavior models formally, proving correctness and generating test cases for each unit;
- Verify and validate the emergent behavior of the communicating components, again generating test cases for integration testing.

Figure 1.1 depicts the role of our framework and its involvement in the V-model.

This work is structured as follows. Chapter 2 presents the concepts on which the proposed framework is built. The theoretical challenges of designing such a framework is described by Chapter 3. Chapter 4 provides more details about implementation challenges, while the description of our case-study demonstrating the capabilities of the approach is presented in Chapter 5. Finally, Chapter 6 provides concluding remarks and ideas for future work.

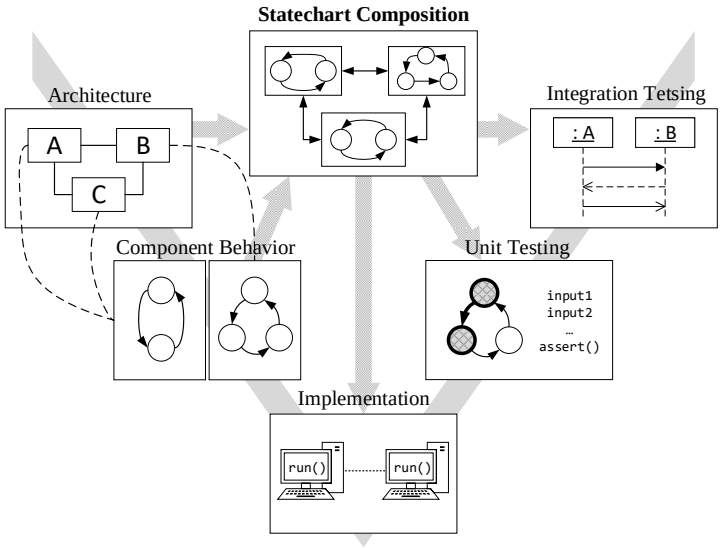

Figure 1.1: The process.

Chapter 2

Background

This chapter introduces architecture description languages and their extensions. This is followed by common component behavior modeling formalisms such as state machines and timed automata are presented, as well as some implementations that have been used in the first prototype of the framework. The chapter ends with the description of the concept of model transformation which serves as the basis of model-driven development.

2.1 Architecture description languages

A large number of architecture description languages are used in industrial and academic environments. In this work the term architectural description language is used for any languages that are used for the specification of dependencies and connections of system components [1]. They are used in the design phase of software development and their goal is to define and analyze software components that will later be the basis of further software engineering activities [7].

Note, that this definition is not too specific, as system design is a complex process with lots of dimensions, perspectives and possibilities. To suit these different perspectives tens of ADLs have been designed with different syntax and semantics, each of them focusing on various aspects of system design. As their features, tool support and system representations vary, their use might affect the remaining software engineering process fundamentally. The use of an ADL that offers the possibility of source code generation, system verification and validation in addition to the basic documenting features might prove invaluable to system architects and significantly reduce the period and cost of later stages.

Different developer teams and architects might have special needs and find different features useful. A recent research has presented an investigation into the industrial applications of architecture languages [12]. Their results unravel the following findings:

- Participants of the survey are usually not satisfied with the analysis features of architectural languages.
- There is room for improvement in terms of semantics, usability, precision and simplicity.

The framework presented in this work has been designed to satisfy these needs as much as possible.

2.2 State machine formalism

State machine is a model of computation to describe the behavior of a system, component or object in an event-driven way [2]. Mathematically, a state machine is a 5-tuple: $SM = \langle I, O, S, s_0, T \rangle$ where:

- I is a finite set of input events or signals that are stimuli from the environment
- O is a finite set of output events or signals that are stimuli for the environment
- $S = \{s_1, s_2, \cdots, s_n\}$ is a finite set of states
- s_0 is the initial state
- $T \subseteq (E \times S) \times (S \times O)$ is a finite set of transitions, that represent changes of states in response to input events and generate output events

There are various extensions to the described state machine formalism that facilitate the compact modeling of hierarchical and concurrent systems. One such extension is statecharts, which also supports auxiliary variables in addition to supporting concurrency and state refinement. In the following subsections two statechart formalisms are presented that allow the description of event driven systems in a high-level, hierarchical way.

2.2.1 Yakindu

Yakindu[1] is a toolkit for the model-driven development of reactive, event-driven embedded systems. It supports the creation of complex hierarchical statecharts. Yakindu provides a graphical editor where the structural elements can be chosen from a palette and instantiated in the view. Variable declarations, actions and transition parameterizations can be specified using a textual notation. To support users in designing well-formed statecharts the tool provides basic validation features. It includes syntactic and semantic checks on the entire model that are live, therefore the users get feedback on their work immediately.

Syntactically correct and validated statecharts can be simulated. Declared events can be raised using a graphical interface and the change of states and variables can be observed in different views. With this feature, basic testing of statecharts can be done at design time.

Yakindu also supports source code generation from syntactically correct and validated statecharts. The generated code presents well-defined interfaces, which hide the details of implementation and provide access only to event raising, variable check and active state check. Code generation can be customized with configuration files specifying the expected features of the generated code.

The following paragraphs present the semantics of Yakindu elements.

Statechart Statechart is the root element of the Yakindu model. It contains one or more top Regions. If there are multiple top Regions, they are orthogonal, i.e. they are assumed not to interfere or communicate with each other in any way. Furthermore, Statecharts may contain multiple Interfaces.

[1] https://itemis.com/en/yakindu/statechart-tools/

Interface Interfaces represent a surface on which Statecharts can be controlled by raising events and their variables can be set and checked. Interfaces contain Event definitions and Variable definitions. Output events are also defined on Interfaces.

Region Region is the container element of the structural elements defined in the following paragraphs. A Region can either be a top Region, contained by a Statechart or a subregion, contained by a composite State.

Entry An Entry node is used for specifying the first active State of a Region. Only one Transition can leave it, the target of which defines the initial State of the particular Region.

Shallow history Shallow history nodes can be placed only into subregions of composite States. They are used to remember the last active state of their parent Regions. If the particular Region is entered, the last active State of the particular Region will be active again. If the Region has not been entered before, the Transition going out of the Shallow history node will specify the active State (same behavior as Entry).

Deep history Deep history is similar to Shallow history, but it affects each nested subregion transitively as well.

State A State represents a stable situation of its parent Region. It can have Entry and Exit events which specify different actions that have to be taken when the state is activated or deactivated, respectively. Furthermore, it can contain Local reactions which are actions that have to be taken if the State is active and the specified events are raised (similar to self-loop Transitions, but without leaving and re-entering the State). *Composite* States extend *simple* States with the ability of containing one ore more Regions. If multiple Regions are contained by a particular State, they are orthogonal.

Choice Choices are syntactic sugar used for splitting Transitions. Each time a Choice is entered, all guards of its outgoing Transitions are evaluated according to specified priority order. If a guard is evaluated to true, the corresponding Transition it fires. Choices are useful for avoiding ”code” duplication (trigger and action specification).

Exit node Exit nodes can be used only in subregions of composite States. These composite States must have an outgoing Transition that does not contain any triggers or a guard. This Transition is called *default Transition*. Whenever the Exit node in the subregion is entered, the subregion is exited and the default Transition fires.

Final state Whenever a Final state is entered in the Yakindu model, the execution stops and the statechart keeps its state until restart, i.e. no Transitions or State events can fire. Checking variables and active States in other orthogonal Regions that have not reached a Final state remains possible.

Transition Transitions specify State changes in a Statechart. They can connect nodes of different Regions, unless these Regions are orthogonal. A Transition must contain a trigger (except for default Transitions of composite States), and can contain a guard and an action. A Transition can fire if its source State is active, the raised event is its trigger, and its guard (if it has one) is evaluated to true. Unguarded Transitions can fire if the corresponding event is raised. A firing Transition executes its assigned action if it has any. This can be either a variable update or the raising of an event.

Figure 2.1 demonstrates the connections and associations of the elements presented in this section.

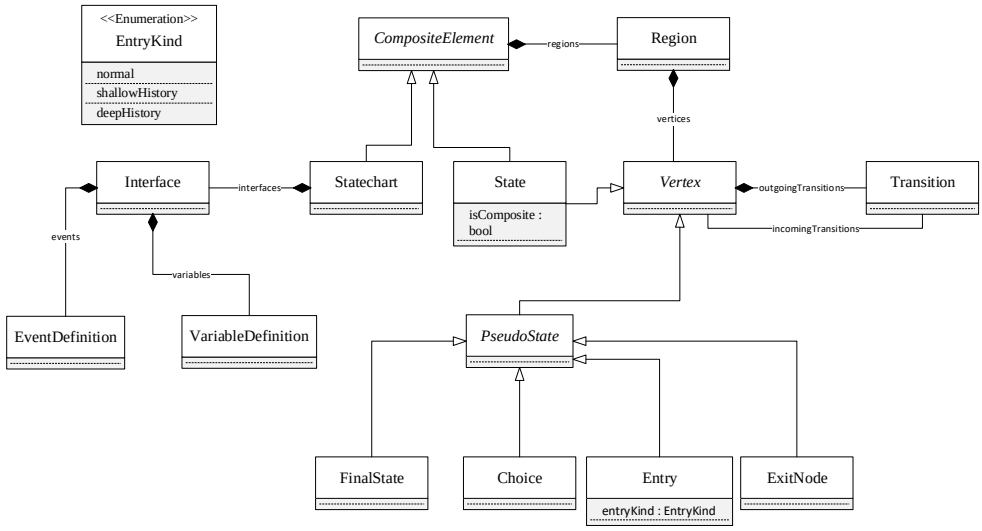

Figure 2.1: The relevant part of the Yakindu metamodel.

2.2.2 Theta

Theta [2] is a configurable and extensible framework for verification of state-based behavior. It is being developed at the Fault Tolerant Systems Research Group at Budapest University of Technology and Economics. It aims to provide formalisms and languages to model software and hardware systems. These formalism are symbolic transition systems, timed automata and control flow automata. In addition to the modeling formalism it provides model-checking algorithms for verification e.g., CEGAR-based reachability analysis.

The following paragraphs presents the symbolic transition system. Table 2.1 presents Theta elements whose semantics equal to Yakindu elements.

Statechart specification This element is the root of all theta statecharts. It contains Constant declarations and Signal Declarations in addition to a Statechart definition.

Statechart definition Statechart definitions contain Variable declarations and one or more Regions. These Regions are top Regions.

State Theta States are very similar to Yakindu States. The only difference is that theta States can not contain Local reactions.

[2]https://github.com/FTSRG/theta

Table 2.1: Theta elements with semantics conforming to Yakindu elements

Yakindu element	Theta element
Region	Region
Transition	Transition
Entry	Initial state
Shallow history	Shallow history state
Deep history	Deep history state
Choice	Choice state

2.3 Timed automaton formalism

Timed automata are an extension of finite automata where a new type of variable is introduced, called clock. Clock variables are real-valued. As clock variables represent the passing of time, their values increase at the same speed during a run of an automaton. Clock variables can be compared to values to be part of transition guards, timing the firing of transitions. Also, clock variables can be reset in transition updates [6].

Real-time systems, such as packet-switched networks, distributed systems and real-time embedded systems can be modeled and analyzed using timed automata.

Mathematically, a timed automaton [5] is a 5-tuple: $TA = \langle I, O, C, S, s_0, T \rangle$, where:

- I is the set of input events, signals that come from the environment and have indicating purposes
- O is the set of output events, signals that can be processed by the environment
- C is the set of clock variables
- $S = s_1, s_2, \cdots, s_n$ is a finite set of control locations, representing abstract situations of timed automata
- s_0 is the initial state of the timed automata
- $T = T_c \cup T_w$
- $T_c \subseteq (I \times S \times C) \times (S \times O \times C)$ is a finite set of classic transitions, that represent changes of states on particular input events and generate output events. Time does not passes during classic transitions.
- T_d is an infinite set of delay transitions, that set the value of clock variables with value C to $C + \tau$, i.e. represent the passing of time, leaving other factors unchanged.

The actual state of a timed automaton is specified by the control location and the current values of its clock variables.

2.3.1 UPPAAL

UPPAAL[3] is a tool for the modeling, simulation, validation and verification of real-time embedded systems. UPPAAL uses the timed automata formalism presented in Section 2.3

[3] http://uppaal.com

with the extension of data types and variables as well as supporting concurrent automata synchronizations through channels.

In the following paragraphs the supported elements of UPPAAL automata are presented [3] [4].

NTA NTA (network of timed automata) is the root element of an UPPAAL system that contains Templates.

Template Templates are automata "types" with parameters that can be bound to values during instantiation. Templates contain Locations and Edges. Each Template contains exactly one initial Location.

Location Locations represent situations of their parent automaton. Locations have one of the following types:

- Normal: Locations of this type conform to the semantics of states of timed automaton.
- Urgent: Urgent Locations represent a situation where time does not pass, i.e. the automaton must fire any enabled *classic* transitions before firing a *delay* transition.
- Committed: Committed Locations are even more restrictive than urgent ones. Not only is time not allowed to pass in them, but restrict the set of enabled Edges to its own outgoing Edges. This means, that committed Locations must be left as soon as possible, prioritizing their outgoing Edges over Edges of other urgent and normal Locations.

In addition, extra behavior can be added to Locations using *invariants*. Location invariants are side-effect free expressions that are evaluated to boolean values. They must always hold while their corresponding locations are active. If the invariant becomes false, the Location must be left at the same time, otherwise a deadlock occurs.

Edge Edges conform to the semantics of transitions of timed automata. An Edge may contain synchronization channels, guards, updates and selection expressions.

Variables UPPAAL extends the timed automata formalism with variables. The following variable types can be instantiated: *int*, *bool*, *clock*. Variables can be used in guards of Transitions, updates of Transitions, invariants of Locations.

Clock Clock variables conform to the semantics of clocks of timed automata. They can be used in guards of Edges and invariants of Locations. They can be reset using updates on Edges.

Synchronization Interactions of asynchronous systems in UPPAAL can be modeled with the use of synchronizations. There are two types of synchronizations in UPPAAL, *simple* and *broadcast*. In both cases, synchronization channels have to be defined, as they are responsible for the delivery of notifications. Synchronization channels can be

connected to an Edge in one of the following two ways. If the channel is connected using a ! operator, the edge will send a notification on firing (active synchronization). On the other hand, if a ? operator is used, the Edge will wait for a notification before firing (passive synchronization).

A *simple* synchronization takes place between automata A_1 and A_2 if and only if both of them are in a Location l_1 and l_2 with at least one outgoing Edge e_1 and e_2 connected to a synchronization channel *ch*. One of the Edges e_1 must be connected to *ch* with a ! operator, e_2 must be connected with a ? operator and its guard must be evaluated to true. If e_1 fires, synchronization takes place between A_1 and A_2 which results in the firing of e_2. If there are more than one enabled Edges leaving l_2 and synchronizing to *ch* in a passive way, only one of them will be selected for synchronization (non-deterministically).

The process of *broadcast* synchronization is similar to that of *simple* channels. *Broadcast* synchronizations take place between one A automaton and zero or more $B_1, B_2, \cdots, B_n$ automata. The process takes place in the same way as specified in the previous paragraph, with the difference that not only one, but every $B_1, B_2, \cdots, B_n$ automata can synchronize to a single channel. Firing of an Edge actively synchronizing to a broadcast channel can take place even if the number of passively synchronizing automata is zero.

2.3.2 Formal verification with UPPAAL

Formal verification is a method for proving or disproving the correctness of a system with mathematical precision. Correctness is checked with respect to certain properties or specifications given by the user. Model checking is a formal verification technique that explore the behavior of the given model exhaustively, i.e. all relevant behaviors of the model are analyzed (contrary to simulation and testing, which only sample behaviors).

UPPAAL uses model checking techniques to verify timed automata [13]. Certain requirements that are expected of the systems can be described with temporal logic expressions. The language supported by UPPAAL is the subset of computation time logic (CTL). CTL is a branching-time logic which means its model of time is a tree-like structure. It starts from a root (the initial state) and each branch represents a possible execution sequence. The nodes of the branches represent the states the system assumes throughout the execution sequence.

An UPPAAL requirement (called query by UPPAAL) consists of a *path quantifier*, a *temporal operator* and a *state expression*. The state expression can be any boolean expression that is valid in UPPAAL. UPPAAL does not support the combination of temporal operators, however, a special class of expressions in the form of `A[](a implies A<>b)` is implemented, denoted by `a --> b`. The possible combination of path quantifiers and temporal operators are as follows:

- `A[]` ϕ: ϕ must hold on all states of all paths of the execution tree
- `A<>` ϕ: ϕ must hold on at least one state of each path of the execution tree
- `E[]` ϕ: ϕ must hold on all states of at least one path of the execution tree
- `E<>` ϕ: ϕ must hold on at least one state of the execution tree
- ϕ `-->` ψ: if ϕ occurs in state s ψ must hold on at least one state of each path of the execution tree starting from state s.

Figure 2.2 depicts the CTL expressions that are accepted by UPPAAL. The filled circles represent system states where the Boolean expression ϕ holds.

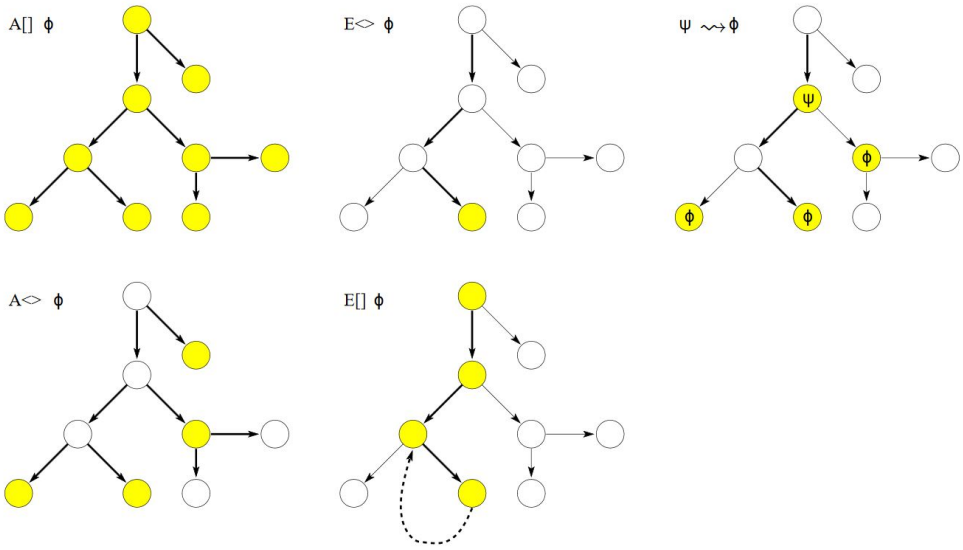

Figure 2.2: Route quantifiers supported by UPPAAL.

State expressions are expressions that can be evaluated on the states of the execution sequence. Their syntax is very similar to transition guards. Furthermore, they must not have side effects. One of the most used expressions are the ones that check whether an automaton is in a certain Location: (*process_name.location_name*). Also, expressions checking variable values can be constructed in the same way as constructing Transition guards.

2.4 Model transformation

One of the main purposes of the framework is to support code generation from various state based languages. In addition, analysis support can be provided by transforming models into formal models, thus formal verification techniques can be applied on them. Conversion between these different modeling formalisms can be achieved by model transformation.

Definition 1 (Model transformation). A process of generating the target model from the source model. The process is described by a *transformation definition* consisting of *transformation rules*, and a *transformation tool* that executes them. A transformation rule is the mapping of elements of the source model to the elements of the target model. [11] ▪

Two factors have to be taken into account in the design of model transformations:

1. Consistency: The source model and the target model must describe the same structure or behavior in their own domains.
2. Traceability: An element in the target model can be traced back to one or more elements of the source model from which it has been generated.

Based on the format of the source and target models, model transformers can be categorized into four groups: model-to-model (M2M), model-to-text (M2T), text-to-model (T2M), text-to-text (T2T).

In case of model-to-model transformations, the source and target formats can be parsed by model-based design tools. During model-text transformation a textual representation is created from the initial format, which can be used by design tools not supporting model based design. Thus, transfer between model-based tools and not model-based tools is achievable.

This work concentrates on graph transformations: graph transformation is the generalization of model to model transformation. Graph transformation is a declarative and formal paradigm that focuses on special rules, called graph-rules. A graph rule has two sides, a left hand side and a right hand side. The left hand side consists of a precondition that specifies when the rule has to be executed, and a graph pattern that has to be matched to a part of the model under transformation. The right hand side also consists of a graph pattern (the elements that have to be created in the target model) and a mapping that associates the elements of these two sides (trace). The elements of the left hand side are fetched using pattern matching. This means the types and attributes of elements of the source model have to be specified in a declarative ways, on which the rule will be applied.

Chapter 3

Theoretical results

The framework presented in this work is designed to support model-driven software development. For the first prototype, we have included support for Yakindu to model state-based behavior as well as to use it for source code generation, and UPPAAL to provide verification and test-generation capabilities.

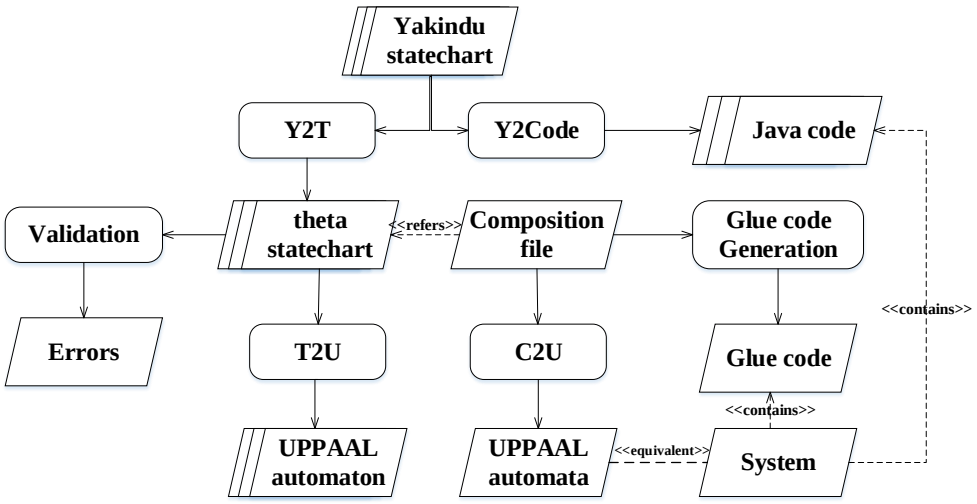

Figure 3.1: The process.

Our proposed workflow to use the framework is as it follows (depicted in Figure 3.1):

- One input of the process is a set of behavioral models (modeled in Yakindu).
- These models are transformed into theta statecharts (Section 3.1 and 4.2) and validated against a set of rules defined in Section 3.2.
- Based on theta statecharts, behavior of the components can be composed using the compositional language defined in Section 3.3.
- Individual statechart as well as their composition can be transformed into UPPAAL automata (Section 3.4) to further validate the model, verify correct behavior and generate test cases for the validation of implementation.
- The last step of the process is source code generation for the composite system reusing the implementations generated by Yakindu (Section 3.5)

3.1 Modeling and code generation: Yakindu

One of the greatest merits of this compositional framework is that it is extensible with arbitrary engineering modeling languages as long as a transformation to the intermediate statechart language is defined. By defining a transformation for Yakindu statecharts, the framework gains the benefits of a rich and easy-to-use engineering tool. Yakindu is a perfect candidate for this role, since its expressiveness and semantics is relatively close to that of theta, and it also provides code generation capabilities for the modeled system.

If more than one engineering domain is supported and can be transformed to the intermediate language, their composition and interaction become possible. Only a trace is needed that contains the mapping of the elements of the different domains, so the interaction of the statecharts in the intermediate language can be back-annotated to the source domain (subject to future work).

Yakindu has a large metamodel with many elements, thus supporting the users in expressing their thoughts and ideas as easily as possible. On the other hand, this huge set of elements raises difficulties when defining a transformation to other languages, since all the elements that are often used by the users have to be mapped to the target language. Therefore, in this work the set of supported features of Yakindu has been restricted to satisfy most needs of users.

The greatest advantage of the intermediate language is that if more than one state-based language is transformable to it, the composition and interaction of models created in them can be simulated and verified, in addition to the independent analysis of such models. Similarly to the transformation of Yakindu statecharts, mapping of other languages, e.g. Papryus, MagicDraw BridgePoint behavioral models to the intermediate language is possible. If such formal mappings of other languages existed, with a little extension of the framework, implementation of their interaction could be automatically generated, in addition to their analysis. Naturally, this would only be possible if the implementations of such engineering models can also be generated.

The transformation of Yakindu elements are mostly straightforward (described in Section 4.2). However, the following elements have no direct equivalent in theta.

Exit nodes The semantics of the Exit node high level construct is transformed with the help of complex submodels in the theta language. Exit node mappings do not appear in the theta model at all, instead each incoming Transition of the Yakindu Exit node is associated with the default Transition of the composite State, i.e. one Transition is created in the theta model whose source is the node equivalent of the source of the incoming Transition of the Exit node, and the target is the node equivalent of the target of the default Transition of the Yakindu composite State.

With this rule, the necessity of Exit nodes are completely eliminated, since this construction enables to get to the target of the default transition of the composite State by firing a single Transition. The triggers, guards and actions of the incoming Transition of the Exit node can be transformed and placed onto the single theta Transition, and the default Transition of the Yakindu composite State can not contain any triggers, guards or actions by definition.

Final States A Yakindu Final state is mapped to a regular theta State. As this construct does not model the semantics of the Yakindu Final state, additional elements and

associations have to be created. As it has been mention in Section 2.2.1, whenever a Final state is entered in the Yakindu model, the execution stops and the statechart keeps its state until restart. This can be achieved with guards placed on each Transition in the theta model referring to a boolean variable "end". The guard contains the negated form of the variable: "!end". This variable is set to true when the State corresponding to the Yakindu Final state is entered (in an entry action).

Time Event specifications of Transitions A Timeout declaration is created in the theta model for all Time Event specification of Yakindu. This Timeout declaration is set to the specified value at each enter of the source state of the Transition. The specified value can be a constant, or an expression. Yakindu expressions described in the Yakindu Expression transformation section can be transformed and used as time values. A Timeout event is created and placed onto the theta Transition equivalent of the Yakindu Transition. Note, that only one unit of measurement (s, ms) can be used in the Yakindu model, otherwise this method leads to unexpected behavior in the theta model.

Local reactions of States Local reactions that are neither Entry events nor Exit events have to be treated differently. The mapping depends on whether the State is composite and it has any Entry or Exit events.

- If the State is simple without Entry or Exit events, a loop Transition is created in the theta model whose source and target is the State equivalent of the Yakindu State. The loop Transition contains all the transformed triggers, guards and actions of the original Yakindu Local reaction.
- If the State is composite or has Entry or Exit events, a subregion (parallel to the other subregions) is created to the State equivalent of the Yakindu State, which contains an Initial state, a simple State and a loop Transition whose source and target is the simple State. The loop Transition contains all the transformed triggers, guards and actions of the original Yakindu Local reaction.

Why is an extra subregion needed in the second case? Considering the presented transformation rules, a loop Transition would not model the behavior of a Yakindo Local reaction correctly. If the Sate has Entry or Exit events, at every fire the Entry and the Exit actions would be executed. If the State is composite, an exit and an entry would take place at every fire, setting the subregions to their initial states.

3.2 Validation of statechart models

Validation rules have been added to the compositional framework in order to support the users and provide them with the most information possible at the earliest stage of statechart-based system design. While some of these rules are only there to warn the user of particular things (these are marked with a warning label), others must hold if a well-behaving, deterministic, transformable and verifiable system is desired (marked with an error label). The validation rules for best practice use are the following:

3.2.1 Warnings

This subsection presents rules that merely warn the users of the possibility of mistakes.

Transitions into parallel regions A transition shall not go into a node of a parallel region. Doing so would set the other regions to their defined initial states, which could cause confusion.

Same triggers in parallel regions raising the same Signal declaration A transition shall not have the same trigger as another transition in a parallel region and raise the same Signal declaration at the same time. This could cause trouble if the Signal declaration has a type. According to the semantics the statecharts would only react to the second event and process that value – which is chosen non-deterministically.

Same triggers in parallel regions with assignments to the same variable A transition shall not have the same trigger as another transition in a parallel region and assignment to the same variable. Otherwise, only the second assignment is enforced.

Occluded transitions An unguarded outgoing transition of a composite state shall not have the same trigger as an unguarded transition in one of its subregions. This way the transition in the higher hierarchy level occludes the other one, making its firing impossible.

Unguarded outgoing transitions with time event specifications Multiple unguarded transitions coming out of the same node shall not have time event specifications. Otherwise, the transition with the lesser time value occludes the other one, making its firing impossible.

Choices with less than two outgoing transitions A choice shall have at least two outgoing transitions.

Unused Signal declarations All of the declared events shall be used in the model.

Unused Variable and Constant declarations All of the declared variables shall be used in the model.

3.2.2 Errors

This subsection presents rules that must hold, if a well-formed, deterministic, transformable and verifiable model is desired.

Transitions without a trigger Every transition must contain a trigger apart from the following ones;

- transitions going out of an entry,
- transitions going out of a choice.

A transition without a trigger is unable to fire.

Unreachable nodes Every non-entry node must have at least one incoming transition. Otherwise, it is not possible to reach them.

Final states with outgoing transitions A final state must not have outgoing transitions. If a final state is reached, the execution stops, so no other transitions can fire (including the transitions going out of a final state).

Entries with incoming transitions An entry node must not have incoming transitions.

Entries with more than one outgoing transitions An entry node must not have more than one outgoing transitions. Otherwise the first stable state would be chosen non-deterministically.

Entry transitions with triggers A transition going out of an entry node must not have a trigger or guard.

Choices with unguarded transitions A choice must have at most one transition without a guard. Apart from default transitions, all outgoing transitions must have guards.

Choices with triggered transitions A choice must not have triggered outgoing transitions. The incoming transitions of the choices should contain the triggers.

Default transitions with guards A transition with a default trigger must not have a guard.

Unguarded transitions with same source and trigger Unguarded transitions coming out of the same node must not contain the same trigger. Otherwise, there is nondeterminism in the system, since all of them are able to fire if the corresponding Signal declaration is raised, and only one of them will.

Regions without an entry Each region must contain an entry. This rule makes sure that each time a region is activated, it gets into a deterministic state.

Regions with more than one entry Regions must not contain more than one entry.

Transitions across parallel regions A transition must not connect nodes of parallel regions.

3.3 Compositional Design

The framework presented in this work enables to define a composition of statecharts with the help of a domain specific language (DSL). This section defines the grammar of the DSL and introduces the semantics the composite system conforms to.

3.3.1 Grammar

A composition of systems in the compositional tool of the framework consists of the following parts:

- Components: A Component refers to a theta Statechart specification. For each Component an Interface has to be defined.
- Interface: An Interface of a Component contains Ports. An Interface does not need to have a Port declaration for each theta Signal declaration defined in the Statechart specification.
- Port: Ports are endpoints the Instances of the particular Component will send or receive messages on. A Port refers to a theta Signal declaration and it also has a direction (IN or OUT). A Port with the direction of IN may only *receive* messages, OUT ports are for *transmitting* messages. A Signal declaration referred to by a Port may have a type. This means values can be transmitted inside messages.
- Instances: Instances can be created from Components. A Component may have multiple instantiations or none at all.
- Channels: Connecting Ports of multiple instances can be done using Channels. A Channel has one or more Inputs and one or more Outputs. An Input consists of an Instance and a Port with the direction of OUT. An Output is similar to an Input, but their Ports are with the direction of IN. Whenever a Channel gets a message through any of its Inputs, the message is sent to each Output, which means all Instances connected to the particular Channel will get the message. Connecting Ports with the same direction is not possible in this tool. Ports referring to Signal declarations with different types may not be connected.
- System Interface: The System under design has an Interface. This Interface consists of zero or more System IN Ports and zero or more System OUT Ports.
- System Ports: An IN/OUT Port of the system is an alias of an IN/OUT Port of one of its Instances that is visible on the Interface of the System. For instance if a message arrives to an IN Port of the system, it will be forwarded to the corresponding Port of the specified Instance instantly (in the same cycle).

For ease of understanding, an example is presented that defines a composition of statecharts using the specified DSL. The system consists of two Components, Alpha and Beta referring to AlphaStatechart and BetaStatechart, respectively. AlphaStatechart has a Signal declaration called *a* as well as BetaStatechart has one, called *b*. Both components are instantiated: alpha and beta. The *a* out port of alpha is connected to the *b* in port of beta. Also, the the *b* port of beta can be raised through the interface of the system. Furthermore, port *a* of alpha is lead-out to the system interface.

```
interface {
    in {
        b : beta.b
    }
    out {
        a : alpha.a
```

```
    }
}

AlphaStatechart Alpha {
    a : OUT a
}

BetaStatechart Beta {
    b : IN b
}

Alpha alpha
Beta beta

channels {
    [alpha.a] -> [beta.b]
}
```

3.3.2 Semantics

Formally, a composition is a 4-tuple: $C = \langle SC, CA, IN, OUT \rangle$ where:

- $SC = \{\langle S_1, s_1^0, T_1, I_1, O_1 \rangle, \cdots, \langle S_n, s_n^0, T_n, I_n, O_n \rangle\}$ is a finite set of state machines, formally defined in Section 2.2.
- $I = \bigcup_{j=1}^{n} I_j$, i.e. the union of all in events of state machine components
- $O = \bigcup_{j=1}^{n} O_j$, i.e. the union of all out events of state machine components
- $CA \subseteq 2^O \times 2^I$, i.e. channel associations relate a finite set of outputs to a finite set of inputs
- $IN \subseteq I$, i.e. the input interface is a subset of the union of the in events of state machine components
- $OUT \subseteq O$, i.e. the output interface is a subset of the union of the out events of state machine components

A sequence of steps $\varrho = (\tau_1, \tau_2, \cdots)$ is called a *complete run* of C if the following conditions hold.

- $\tau_j = (\underline{s}_j, i_j, \underline{s}'_j, o_j)$ is a single step that consists of a state vector representing each state of each component before the step, a finite set of inputs, a state vector representing each state of each component after the step and a finite set of outputs generated by the state machine components, where at least one of the following conditions holds:
  - $\forall 1 \leq k \leq n : (\underline{s}_j[k], i_j, \underline{s}'_j[k], o_j) \in T_j$, i.e. if a transition is defined in a state machine component that is triggered by the input set, then the transition fires taking the state machine to its target state and producing the corresponding outputs;

- $(\underline{s}_j[k] = \underline{s}'_j[k] \wedge o = \emptyset \wedge \nexists s', o' : (\underline{s}_j[k], i_j, s', o') \in T_j)$, i.e. a component is allowed to do nothing if and only if it has no transition that is triggered by input i_j in state $\underline{s}_j[k]$.

- $\underline{s}_1 = (s_1^0, s_2^0, \cdots, s_n^0,)$, i.e. at the beginning of the run, all state machine components are in their initial states

- $\underline{s}'_j = \underline{s}_{j+1}$, i.e. the state vector at the end of a step and at the beginning of the next step are equal

- $o_j \subseteq i_{j+1} \subseteq o_j \cup IN$, i.e. the inputs of a step is at least the outputs of the previous step and maybe some additional events of the input interface

- ϱ is either infinite or the following condition holds:

  - $\nexists (o, i) \in CA : o \cap o_n \neq \emptyset$, i.e. the execution of steps can only be stopped if the last step does not produce any outputs that can be processed in the next step as inputs

A partial run of a composite system can be any prefix of a complete run (any other sequence is not considered to be a behavior of the composite system).

The compositional tool has a port system that differs from most methods of tools of this field. Each Port of each Instance contains two cells, an outer and an inner cell. Whenever a message arrives at one of the ports, it is placed into the outer cell. If more than one message arrives at the same Port in the same turn, only the latter one is stored removing the former message from the outer cell. [1]

The compositional tool adopts a turn-based semantic. At each turn, the values of the outer cells are copied into the inner cells and the outer cells are cleared. After that, a scheduling turn begins. A Component instance only takes notice of messages in the inner Port cells. The Component instances are scheduled one after another. Although, the order of the scheduling of the instance is not defined, it is fixed, therefore they are scheduled in the same order in each turn. This does not cause a loss of generality, because the instances can not affect each other in one scheduling turn, as an incoming message is placed into the outer cell.

3.4 Verification and test generation: UPPAAL

This compositional framework is extensible not only with engineering modeling languages but also formal modeling languages. A formal modeling language can be used not only for analyzing a single theta statechart, but for verifying a composition of theta statecharts as well. The basics are the same for these two use cases, they only differ at well specified points and therefore have a slightly different semantics.

In this section the rules responsible for mapping the elements of a *single theta statechart* are presented:

[1] This is relevant when the particular Port refers to a Signal declaration with a type. In this case unique values are stored inside messages.

3.4.1 Variable and signal declarations

Variable and constant declarations The following theta types can be transformed: integer and boolean.

Table 3.1: Theta and UPPAAL variable type mappings

Theta type	UPPAAL type
integer	int
boolean	bool

Other theta types, e.g.: real can not be transformed. Constant declarations get a const prefix in the UPPAAL model and they must have an initial value. Variable declarations can also have initial values.

Signal declarations All theta Signal declarations are transformed to UPPAAL broadcast channels. If the Signal declaration has a type, i.e. it can be raised with a value, another variable is created that will be able to store the value of the signal. The following value types are supported: integer and boolean.

This rule makes it possible to follow the semantics of theta statecharts: raising a signal will have effect on each orthogonal regions where a transition with the particular signal trigger is enabled. Using broadcast channels for Signal raising enables templates to communicate with other active "orthogonal" templates, since all active templates will be able to synchronize on the same channel, raise some updates and step to the next location.

3.4.2 Static elements

Regions Each theta Region is transformed to an UPPAAL template, including subregions of composite states. This way the hierarchy levels of the theta model are kept in the UPPAAL model too, ensuring that no information about them (e.g. child states of composite states) is lost during the transformation. Since all regions are represented by separate automata, a method has to be given for distinguishing active and inactive templates, as not all of them are active all the time. For example if a template is the equivalent of a subregion of a composite state, then it is active only if the location equivalent of the composite state is active. To ensure this, every template is associated with a bool variable called isActive to make sure that only edges of active templates can fire. These variables are set to true if the corresponding state is entered, and set to false if the state is exited. These variables are placed onto every edge in a template as a guard.

Entry states Each theta Entry state, including history indicators is transformed to a *committed* UPPAAL location. These locations are set as initial locations of their templates.

States

1. Simple States: Each simple theta State is transformed to an UPPAAL location.
2. Composite States: Each theta compsoite State is transformed to an UPPAAL location l. Moreover, they are given a committed "entry location" el and an "entry

edge" *ee* with synchronizations that connects *el* to *l*. Every incoming transition of the composite state is mapped to an edge $ie_1, ie_2, \cdots, ie_n$ whose target is *el*. This mapping ensures that every time an incoming edge ie_i is taken, all the subtemplates are brought to their proper stages. This stage depends on the Entry node of the corresponding Region of the template. If it is an Initial state, the template gets to its corresponding *committed* initial location. If it has a history, the last active location will be active. In both cases the variable indicating the activeness *isActive* of the template is set to true. If a location of a composite State is exited, then all the subtemplates must be deactivated: *isActive* variable set to false.

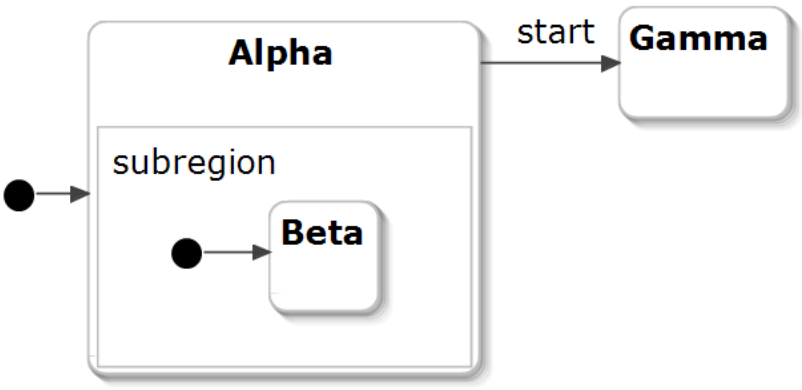

Figure 3.2: A theta statechart with a composite state

Figure 3.3 depicts a theta statechart with a composite State. Figure 3.3 and 3.4 depict the template equivalents of the *main region* and the *subregion* of the statechart, respectively. Note the entry location, called *entryOfAlpha* and synchronization channel placed onto the entry edge of location Alpha. Also, it is important to note the passive synchronization channels *entryChanOfAlpha?* as well as the assignments to variable *isActive* on edges that lead to location *EntryLocation1* from all stable locations of the template. These are the edges that take the template to its proper stage on entry. Finally, note the edges containing synchronizations to channel *exitChanOfAlpha*. These are the edges that disable the template equivalent of subregion (*isActive* variable set to false) on leaving the location of the composite State.

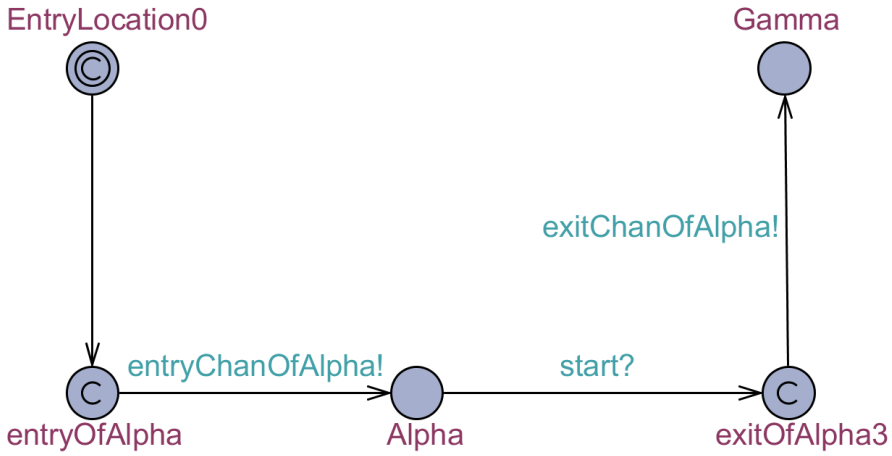

Figure 3.3: The UPPAAL template equivalent of the main region of the statechart depicted in Figure 3.2

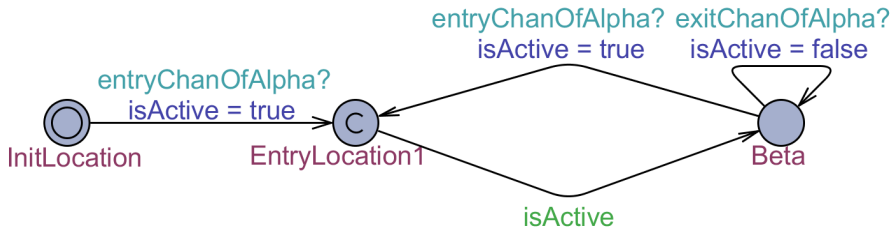

Figure 3.4: The UPPAAL template equivalent of the subregion of the statechart depicted in Figure 3.2

Choice States The semantics of the choice high level construct is implemented with the help of complex submodels in the UPPAAL language. This means, that an edge is created in the UPPAAL model for each incoming-outgoing Transition pair.

Transitions Transitions are transformed differently according to their role in the Yakindu model.

1. Transitions connecting nodes where the source and the target are in the same Region: These kinds of transitions are transformed to UPPAAL edges connecting the Uppaal equivalents of the source and target.

2. Transitions connecting nodes where the source and the target are not in the same Region: This kind of transformation works only if one of the ancestors of the source/-target is in the same region of the target/source. In these cases transitions are transformed to multiple UPPAAL edges with synchronizations ensuring that all of the intermediate UPPAAL templates are brought to their initial stages.

3.4.3 Dynamic expression

Update expressions of Transitions Theta constants and composite arithmetic expressions can be assigned to variables with types specified in Table 3.1.

Guard expressions of Transitions Theta boolean variables and composite boolean expressions can be placed onto Transitions as guards.

Signal actions of Transitions Theta Signal actions are transformed to UPPAAL active channel synchronizations (!). While a theta Transition can contain more than one Signal action, an UPPAAL edge can contain only one synchronization. For this reason, a set of Signal actions of a Transition are transformed to a series of committed locations connected by edges with the corresponding synchronizations on them.

Signal events The Uppaal synchronization channel of the Signal event is placed onto the edge equivalent of the Transition containing the particular Signal event as a passive channel synchronization (?).

Timeout events A clock variable is created for all templates of the generated UPPAAL model. Theta Timeout events are transformed using location invariants and guards refer-

ring to the clock variable of the particular template. For example an "after 1 s" expression on a Transition is transformed to:

- "Timer <= 1" location invariant in the source of the edge equivalent of the Transition
- "Timer >= 1" guard on the edge equivalent of the Transition

Default events Default events are not mapped explicitly. Instead, all Transitions coming out of the same node as the default Transition are selected. Their guards are transformed, negated and their conjunction is placed onto the "default edge".

Entry actions of States For each simple State with an Entry event an entry location and an entry edge that connects the location to its UPPAAL location equivalent are created. For composite States no additional entry locations and edges are created. All the UPPAAL equivalents of the incoming Transitions of the State with Entry event are targeted to the entry location. The Entry action can be one of the followings:

- Variable update: The transformed update is placed onto the entry edge, similarly to the method described in paragraph Update expressions of Transitions.
- Signal action: A set of Signal actions (according to the number of raised Signal declarations) of a Transition are transformed to a series of committed locations connected by edges with the corresponding synchronizations on them. This sequence is inserted between the entry location and the entry edge.

With these methods, it is ensured that every time an UPPAAL equivalent of a State with an Entry event is entered, the necessary actions take place. If the State is composite and there are Transitions going into its substates, the synchronization mechanism takes it into account and the synchronization edges will contain the necessary actions.

Exit actions of States Theta expressions described in the theta Expression transformation section can be transformed. Similarly to Entry actions, these can be one of the following actions:

- Variable update: The transformed update is placed onto each outgoing edge of the UPPAAL equivalents of the particular State.
- Signal action: A set of Signal actions (according to the number of raised Signal declarations) of a Transition are transformed to a series of committed locations connected by edges with the corresponding synchronizations on them. This sequence is inserted right after each outgoing edge of the particular location.

Transitions might go out of substates of composite State into other regions, leaving the composite State as well. The Exit actions are taken in these cases too. The edges that are responsible for the mapping across-regions Transitions contain the necessary expressions.

3.4.4 Formally representing incoming signals

Engineering tools often enable the users to simulate the designed models. Although, this is not the case with theta UPPAAL provides us with this opportunity. While it is possible

for UPPAAL models to be simulated without external signals, theta statecharts model reactive systems. This means that statecharts wait for incoming signals and they react to them according to their current states. This phenomena has to be modeled in UPPAAL if a reactive system semantics is wanted. As the act of sending signals is parallel to the run of the automaton, this model is based on a separate template called Control Template. By default, the template contains a single location, since its state does not depend either on the parallel templates, or the signals that have previously been sent. Furthermore, it contains self-loop edges; one edge for each Signal declaration coming out and going to the single location, so they can be fired one after another. Each edge contains a different UPPAAL active channel synchronizations (!).

If there are Signal declarations with types in the theta model, the Control Template is generated differently. Before raising the synchronizations, their values have to be set properly. To implement this functionality, extra elements have to be added to the template. This is achieved by creating a committed location and a new edge with the necessary update connected from the initial location to the committed location. Next, the committed location is set as the source of the synchronization edge. There can be more update edges from the initial location to the same committed location. For each comparison where a particular signal is compared to an expression an update edge is created. The UPPAAL equivalent of these expressions are placed onto these update edges.

3.4.5 Transformation of composite statecharts

Transforming a composition of statecharts is based on the transformation of a single statechart, but differs from it in the following points:

- Instead of a single statechart, multiple instances of different statecharts have to be transformed. This means that each automaton instance has to have its own templates, locations, edges, variables and signals that are independent of the elements of other instances. This can not be achieved by the utilization of template instantiations. The reason is all instances must have access to variables of other instances and this can not be done if the variables of a instance are bound.

  Instead, the transformation is specified by the extension of all rules defined in Section 3.4: instead of creating a simple element as an UPPAAL equivalent of the theta element, they create multiple elements, one for each instance defined in the composition model.

- According to the semantics defined in Section 3.3.2, each Signal declaration is mapped to two boolean variables instead of broadcast channels (for each instance). The first variable ”toRaise” can be set to true by the user through the Control Template or by other automata indicating that the signal has been raised. At the start of each turn, its value is copied into the second variable ”isRaised”. In the models this variable is used as guards on edges, representing the theta signal triggers.

- Scheduling of the automata has to be realized according to the semantics defined in Section 3.3.2.

  The scheduling is realized with the help of a Scheduler template. Its job is to run each template of each instance in the same order every time a cycle is initiated. The running of a template is realized with the use of ”runCycle” broadcast channels. One such channel is created for each template. This channel is placed onto each edge representing triggered Transitions of the particular template. The scheduling of a

template means firing an edge of the Scheduler template with an UPPAAL active channel synchronization (!) ”runCycle” on it, and let the template instance of an automaton synchronize to it.

The ordering of templates has some restrictions. First of all, the templates of a particular instance have to be scheduled one after another. Moreover, the templates have to be scheduled from the one representing the highest abstraction level to the one representing the lowest one. If there are more templates on the same abstraction level (representing orthogonal regions), their ordering is undefined. Special care has to be taken, when dealing with composite States. Entering the location equivalent of a composite State can only be done at the end of the scheduling of an instance. Otherwise, the subtemplates that would be activated on enter, could take actions, which does not conform to the turn based semantics. To avoid this, a broadcast channel ”finalize” is introduced, which is fired when all the templates of the particular instance have been scheduled. On this synchronization all the templates of the instance get to their stable stage, i.e. locations representing Composite states are entered and its subtemplates are activated.

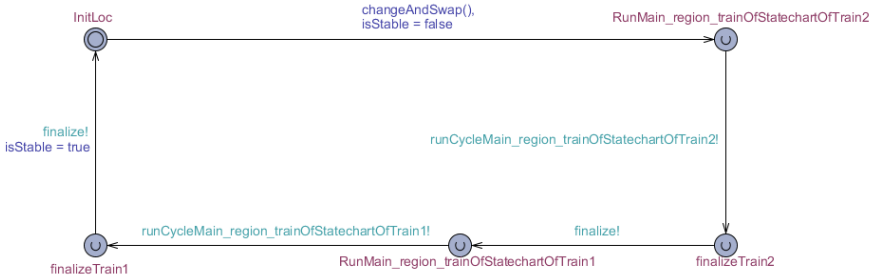

Figure 3.5: A control template of two automata

Figure 3.5 depicts the control template of two automata: *Train1* and *Train2*. As you can see, the instances are scheduled in the order of Train2 and Train1. Note the ”finalize” synchronization at the end of the scheduling of an instance.

- Interactions of the automata has to be realized according to the semantics defined in Section 3.3.2. This is fulfilled as the consequence of the modified Signal declaration mapping and the scheduling process.

- Transitions with Timeout events are mapped to two sequential edges (there is a location between the two of them). The source location of the first edge contains the location invariant and the edge contains the guard referring to the clock. The second edge will contain the necessary ”runCycle” synchronization and the actions if there are any on the theta Transition. This is realization considers the fact, that edges containing UPPAAL channel synchronizations can not contain guards referring to clocks.

3.4.6 Query generation

During the theta-UPPAAL transformation process queries are also generated in addition to UPPAAL automata, which are useful for basic verification of statecharts. The first query can be used to checked whether there is a deadlock in the system. All the other queries check location reachability. Reachability of each state is a basic requirement of all statecharts, and it is hard to check whether a statechart satisfies this criteria and the

process is rarely supported by tools [9]. To address this problem, a query is generated for each state in the theta model that checks whether the location equivalent of the particular state is reachable in the generated automaton. With these queries the reachability analysis is easily executable.

3.5 Code generation

Composite statechart systems can be transformed not only to UPPAAL, but to source code as well. An interface is generated for each component with methods associated to its ports. Furthermore, the Wrapper design pattern is utilized, i.e. each component is wrapped into a class implementing the generated interface. Also, the wrapper class implements the double cell port system presented in Section 3.3.2. Two queues are defined: one of them, called *currentQueue* is responsible for the storage of incoming messages in a particular cycle, the other one, called *lastCycleQueue* stores the incoming messages of the last cycle. In a particular cycle messages of the last cycle are processed, and the new ones are placed into *currentQueue*. At each cycle start, messages of currentQueue are loaded into lastCycleQueue to be processed. In addition, the wrapper class defines methods which are useful for providing information about the component, i.e. its current state.

Generated classes of components should not be used separately. Instead another class, called Container is generated that contains all defined instances of components (Wrapper design pattern once again). Also, an interface is generated that conforms to the interface of the composite statechart system which is implemented by the Container class. Each method is associated to an instance port on which messages can be sent. Furthermore, the Container class is responsible for the connection of the component instances through Listener implementations. For each out port a listener method is implemented that raises in ports, i.e. calls the corresponding methods of the corresponding instances. In addition, Container implements a method called, *runFullCycle* that is responsible for the execution of cycles until all currentQueues of instances are empty.

Chapter 4

Implementation

The number of qualitative modeling tools supporting model-driven driven development are increasing that facilitate implementation. This section introduces the technologies that have been used throughout the development of the framework. In addition, straightforward mapping rules are presented that have been used in the Yakindu-theta transformation.

4.1 Technologies

During the implementation big focus has been put onto fresh and open-source technologies. The development of the framework was carried out on Eclipse environment which inferred the use of Eclipse Modeling Framework, VIATRA transformation framework, and the Xtext framework for language development.

4.1.1 Eclipse environment

Eclipse[1] is an open source platform independent integrated development environment (IDE). It consists of a base workspace (the basis of all Eclipse distributions) and a plug-in system. The latter enables the customization of the environment for example with the EMF Modeling Tools, Yakindu or our own plug-ins.

The framework presented in this work is implemented as a collection of Eclipse plug-ins. These plug-ins are not independent of each other but there are interaction between them. Figure 4.1 depicts the architecture of the framework, presenting the plug-ins and their dependencies.

4.1.2 VIATRA framework

VIATRA[2] is an Eclipse project that supports the development of model transformations with a large variety of tools. These following VIATRA tools are utilized in the development of the framework.

Most importantly, VIATRA offers a language that supports the definition of graph patterns over EMF models in a declarative way [15]. Since statecharts can be regarded as graphs where nodes are elements and their associations are edges, they can be efficiently examined

[1] https://eclipse.org
[2] https://wiki.eclipse.org/VIATRA

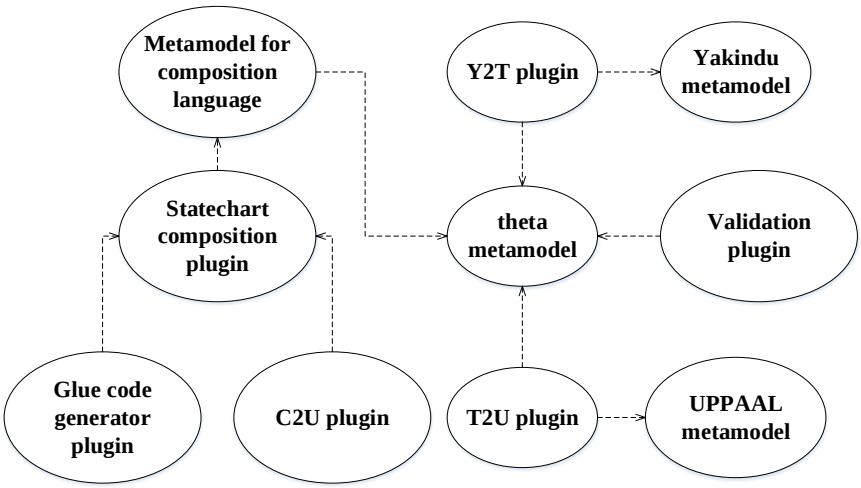

Figure 4.1: The plug-in dependencies of the framework.

as graphs. Graph patterns over statecharts can be used for the selection of model elements to be transformed. Only an appropriate pattern has to be defined that describes the attributes and associations of the required elements. This can all be done in a declarative way, so the user has to focus only on the types of elements, associations and the value of their attributes, the query and the model transformation are taken care of by VIATRA.

Graph patterns can also be used for defining well-formedness constraints and error patterns. This can be done by the utilization of the VIATRA Validation Framework[3]. These patterns show information about the structure of the created models. There are many constraints that can not be specified in the metamodel only (i.e. the number of initial states in a region), therefore they have to be checked in another way. The static analysis process of checking whether these well-formedness constraints hold is called validation. With these means, the users can be given feedback on the correctness of their models at design time, making the design process as productive as possible.

One of the greatest advantages of the VIATRA framework is that the defined graph patterns are evaluated incrementally. This means the traversal of the entire model is done only once, during the first evaluation of graph patterns. After that, the matches of patterns are maintained and modifications of the model will result only in the traversal of the changed part, not the whole model. This method enables the fast reaction to model modifications and facilitates giving feedback to the user in real time.

In addition, VIATRA still offers a framework that facilitates the definition of model transformation rules. For each rule a pattern is needed that returns the elements whose mapping is to be executed. In the rule the new elements in the target domain can be created and the traceability information associating the elements in the different domains can be saved.

[3]https://wiki.eclipse.org/VIATRA/Addon/Validation

4.1.3 XText framework

Xtext[4] is an open-source Eclipse framework for the development of programming languages and domain-specific languages (DSLs). Languages can be specified using a grammar language. Xtext is based on the EMF project: metamodels of the defined languages are Ecore models which can be automatically generated from the defined grammar, or can be manually given. In addition, Xtext provides several features to support development on the specified language: a parser, a linker, a compiler, as well as a typechecker and editing support for Eclipse (syntax coloring, code completion, etc.).

The compositional language of the framework has been created utilizing the Xtext framework. The metamodel of the modeling language was created manually.

Xtend[5] is a general-purpose very high-level statically typed object-oriented programming language that is built in Xtext. It is compiled to Java code, therefore it can be integrated with all existing Java libraries. Also, it has its roots in Java syntactically as well as semantically. However, it concentrates on a tighter, more solid syntax. Additionally, Xtend offers some additional functionality, for instance operator overloading, dispatch methods and extension methods. In addition to object-oriented features, Xtend integrates traits of functional programing, such as lambda expressions.

The implementations of model transformation have been created using the Xtend language. Unique features such as extension methods, dispatch methods and lambda expressions have been extensively used. Therefore, the code has remained readable and concise.

4.2 The implementation of Y-T rules

VIATRA framework facilitates the implementation of straightforward transformation rules. This section presents the mapping of Yakindu elements that have semantics equivalents in theta.

4.2.1 Variable and Event definitions

Variable definitions The following Yakindu types can be transformed: integer, real, boolean and string.

Table 4.1: Yakindu and theta variable type mappings

Yakindu type	Theta type
integer	integer
real	real
boolean	boolean
string	integer

Strings are handled as atoms. Strings used in the Yakindu model are traversed and each one of them gets an *id*. A reference to a particular string is replaced with its id in the theta model.

[4]

[5] http://eclipse.org/xtend/

Variables declared constant in the Yakindu model are mapped to Constant declarations. Otherwise a Variable declaration is created. Yakindu constants must have an initial value. Non-constant variables can also have initial values.

Event definitions Yakindu Event definitions are transformed to theta Signal declarations. If the Event definition has a type, a Parameter declaration is created for the Signal declaration with the necessary type. The supported types can be found in Table 4.1.

4.2.2 Static elements

Regions Each Yakindu Region is transformed to a theta Region.

Entry states Each Yakindu Entry is mapped to a theta Initial state or a Shallow history state or a Deep history state according to the kind of the entry (normal, shallow history or deep history).

States

1. Simple States: Each simple Yakindu State is transformed to a theta Simple State.
2. Composite States: Each composite Yakindu State is transformed to a composite theta State. Naturally, the subregions of the Yakindu State are mapped to be the subregions of the theta State.

Choices Each Yakindu Choice is transformed to a theta Choice state.

Transitions Each Yakindu Transition is transformed to a theta Transition.

4.2.3 Dynamic elements

Assignment expressions of Transitions Yakindu constants and composite arithmetic expressions can be assigned to variables with types specified in Table 4.1.

Guard expressions of Transitions Yakindu boolean variables and composite boolean expressions can be placed onto Transitions as guards.

Event raising expressions of Transitions Yakindu Event raising expressions are mapped to Signal actions referring to the Signal declaration equivalent of the Yakindu Event and placed onto the theta Transitions as guards.

Regular Event specifications of Transitions Regular Event specifications are mapped to Signal events referring to the Signal equivalent of the Yakindu event.

Default triggers of Transitions Defualt triggers are mapped to Default events and placed onto the theta Transition equivalent of Yakindu Transition.

Entry events of States Yakindu Entry events are mapped to theta Entry actions. The Reaction effect of the Yakindu Entry event can be the following two: Assignment Expression or Event raising expression. They are transformed as it has been defined in Paragraph Assignment expressions of Transitions and Event raising expressions of Transitions, the difference is the created expressions are referred from an Entry action and not placed onto a Transition.

Exit events of States Yakindu Exit events are mapped to theta Exit actions analogously to Entry events.

Chapter 5

Results

In this chapter the application of the framework is demonstrated on a real case-study of the critical cyber-physical system domain. The selected case-study is the so-called MoDeS^3[1], which is based on a railway transportation system controlled by the users in which a distributed safety logic prevents the collision of trains.

5.1 MoDeS^3 case study

The goal of the MoDeS^3 project is to apply model-based development techniques, open source modeling and various verification techniques in the development of distributed safety critical systems.

5.1.1 Introduction of MoDeS3

The demonstrator is a railway system. Multiple trains move on tracks which are built of sections and turnouts. The trains are controlled by the user by changing their travel direction and speed. Turnouts are also controlled by the user: the direction can be switched so different paths of the railway can be traversed by the trains. The positions of trains are detected by sensors embedded into the tracks: they sense the trains and notify the corresponding *embedded computer*. Each embedded computer is connected to local sensors of the sections and turnouts, and are responsible for gathering all the information that these components have to offer. There are six BeagleBone Black (BBB) embedded computers altogether in the demonstrator, serving as the controllers of the specified track components. Components belonging to a single BBB are called a *zone*. Note, that these BBBs can have only local information which means they have to cooperate and prevent accidents. This makes MoDeS^3 a distributed system.

5.1.2 Interlocking safety logic

MoDeS^3 is a distributed system, as BBBs only have direct information about their own zone. This hampers accident prevention, since information has to be gathered in one zone, which then has to be delivered to controllers of adjacent zones via network. As incidental packet losses might have critical consequences, the distribution of information has to be supported by a reliable protocol that handles this phenomenon. To conclude, safety has

[1] http://modes3.tumblr.com

to be assured inside a single zone, as well as on the edges of zones. The latter has to be supported by a secure information distribution protocol.

The safety system uses statecharts to prevent the collision of trains. Two different statecharts have been designed, one to describe the behavior of a single section, and another one to describe the behavior of a single turnout. These statecharts have been designed in Yakindu. A single statechart is associated to each element of a zone corresponding to its type. The statecharts of a zone are composed, thus modeling the behavior of an entire zone. Source code can be generated from these compositions, which then can be deployed onto the BBBs. The interaction of BBBs are implemented using the MQTT protocol.

A section has been abstracted to focus only on its qualities that are relevant in the safety logic. As a section has two endpoints, the model concentrates on two directions it can receive or send notifications to. This has been modeled by the use of in-events and out-events, one for each notification type and one for each direction. The following notification types used in inter-section communication are present in the model:

- Reserve: This is sent to adjacent sections from a section occupied by a train.
- CanGo: This is the *positive* answer to a reserve notification if the section is *free*, i.e. the train *can* proceed onto the particular section.
- CannotGo: This is the *negative* answer to a reserve notification if the section is *not free*, i.e. the train *can not* proceed onto the particular section.
- Release: This is sent to adjacent sections from a section that has just been left by a train.

The following notification types are used between the section and its corresponding controller, holding information about the arrival and the leaving of a train:

- Occupy: This notification is sent to a section if it has been occupied by a train.
- Unoccupy: This notification is sent to a section if a train has left it.
- Stop: This notification is sent to the corresponding controller if the section wants to stop the train standing on it.
- RestartProtocol: This notification is used for reseting states in state Stop.

Sections not affected by any trains are in state Free. If a train is placed onto a section b the section goes to state Occupied. The adjacent sections of the occupied section a and c go to state Reserved. The sections in state Reserved and in state Occupied (a, b and c) are said to be in the ”aura” of the particular train. If the train moves from section b and reaches the adjacent one c, c tries to reserve the other adjacent section d. If the reservation proves to be successful, c and d go to states Occupied and Reserved, respectively. It is important to note, that section b stays in state Occupied. When section b is left entirely by the train, it goes to state Reserved, while releasing a which goes to state Free.

In the previous example section c successfully reserved section d. On the other hand, if the reservation had failed, section c would have gone to State Stop. If the sections in state Reserved get any more reservation claims by sections not belonging to their own ”aura”, they will respond with a *CannotGo* notification. This will be processed by the section sending the reservation claim, causing it to go to State Stop which involves sending a

stop notification to the train standing on it. State Stop also sends *CannotGo* notifications in response to incoming reservation claims, which ensures to stop both of the trains proceeding towards one other.

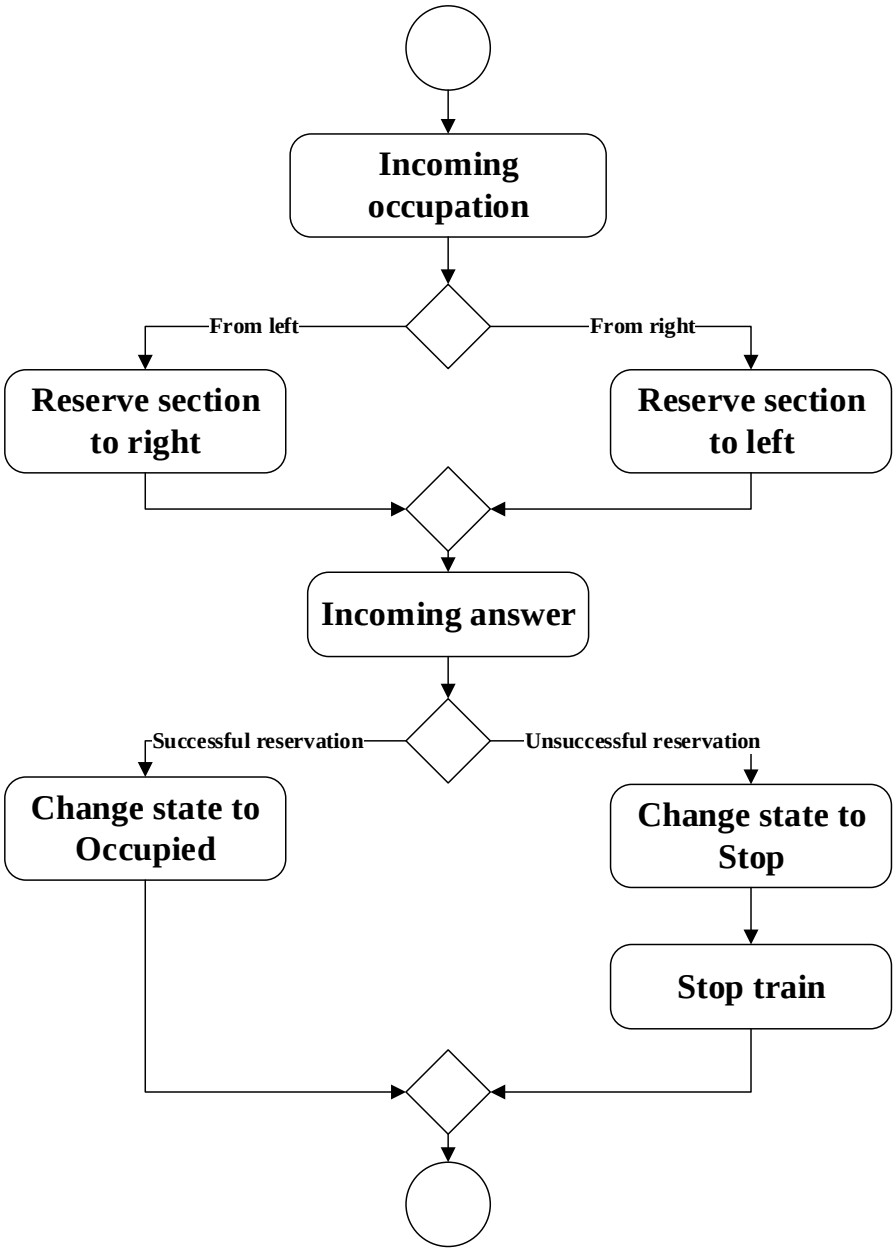

Figure 5.1: The behavior of a section when it gets an occupation notification.

Figure 5.1 demonstrates the behavior of section c when it is reached by the train.

Note, that this algorithm prevents the collision of trains going into the same direction in addition to trains proceeding towards each other. In the former situation only the back train is stopped, the other one may continue its way.

Sections in state Stop can be reset in one of the following ways. A RestartProtocol notification can be sent to them, on which they try to reestablish the ”aura” of the stopped train. Also, stopped trains can be removed from sections manually. This results in sending

an Unoccupy notification to the section in state Stop, on which it goes into state Free while sending Release notifications to adjacent sections.

The turnout statechart has been designed similarly to section statechart. It supports each notification type used in inter-section communication. Furthermore, it supports a Switch notification in addition to Occupy and Unoccupy. Since trains must not stop on turnouts, Stop and RestartProtocol notification are not supported.

Notifications of inter-section communication are not directly processed by turnout statecharts, but are passed to an adjacent section statechart. The particular section depends on the state of the turnout, whether it is Straight or Divergent. The adjacent section processes the notification and if it has any responses, they are transmitted back to the section initiating the notification exchange.

Switch notifications can be sent to turnouts at any time but they will only change their states if there is no train standing on them. However, situations where a user switches a turnout while a train proceeds towards one of its endpoints must be addressed. In these situations, the endpoint gets locked-up, the switch of the turnout state (Straight to Divergent or vice versa) is made, and the section under the train is put into state Stop.

5.1.3 Supporting the development and the verification

The framework presented in this work was used in multiple phases of the development of the interlocking safety logic. The validation possibilities were used during the design of the statecharts in addition to basic Yakindu validation. Some validation rules can not be checked by Yakindu, such as non-deterministic behavior and occlusion of transitions. These were discovered with the use of the validation plugin of our framework well before the simulation and testing of the models even began. After the statecharts have been finished, UPPAAL automata were generated from the statecharts, and reachability and deadlock freedom criteria were checked.

As the interlocking system is based on the interaction of statechart instances, the composition of them had to be constructed according to the design of the track. This was done using the compositional language. The corresponding ports of the statechart components were connected, which was followed by the generation of source code. Source code could be deployed onto the BBB controllers that are responsible for the interactions of components belonging to their particular track part.

5.1.4 Formal verification of the safety logic

Analysis can focus on the interaction of elements in a single zone as well as the interaction of multiple BBBs. The latter one is based on the implemented network protocol, therefore this work presents the former one.

Dangerous situations in a single zone can show up in the following ways: *1)* trains proceeding towards each other leading to collision and *2)* one train going into another from the back. These situations are detected by two independent safety systems.

Sections interact with each other as they sense the arrival and the leaving of a train. To verify their emergent behavior a *train model* has to be created. The model contains three states which represents the position of the train. State T1 represents its initial position, the section it is placed onto manually. Reaching State T1T2 means the train has reached the next adjacent section but has not left the initial one entirely. State T2 represents the

train completely leaving its initial section and fully taking the adjacent one. The states can be changed with the raising of the *move* event, which symbolizes the proceeding of the train. Also, there is a boolean variable *disabled* which can be set to true (by sections), thus preventing further movement. This model enables one-way proceeding of trains only.

A composition has been created to model the railway system. The track consists of the sequence of six sections. They are called *section1*, *section2*, ..., *section6*. The sections are connected in a way that enables them to correctly interact with one other, i.e. they are able to send and receive Reserve and Release as well as CanGo and CannotGo notifications to/on the correct ports, thus implementing the safety logic. The first section of the sequence can only interact with the second one, and the sixth section can only interact with the fifth one. Two trains, *train1* and *train2* are instantiated and placed onto *section2* and *section5*. The train instances have to be connected to sections, so trains can notify sections of their positions (Occupy and Unoccupy) and sections are enabled to stop trains (Stop). This can be done in two separate modes each modeling one of the dangerous situations described at the beginning of the section:

1. *Train1* is connected to *section2* and *section3*, *train2* is connected to *section5* and *section4*. *Section2* and *section5* are represented by state T1 in the train model, while *section3* and *section4* are represented by T2. This layout models two trains proceeding towards each other as it can be seen in Figure 5.2.

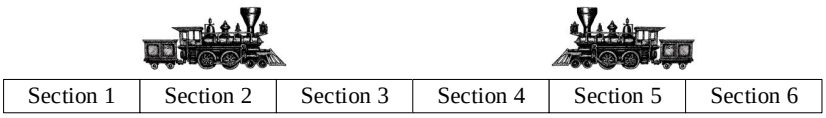

Figure 5.2: The layout where two trains proceed towards each other.

2. *Train1* is connected to *section2* and *section3*, *train2* is connected to *section5* and *section6*. *Section2* and *section5* are represented by state T1 in the train model, while *section3* and *section6* are represented by T2. This layout models one train proceeding towards another one from the back as it can be seen in Figure 5.3.

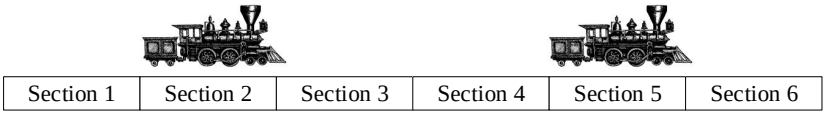

Figure 5.3: The layout where a train proceeds towards another one from the back.

Two separate compositions have to be constructed in these ways, so both dangerous situations can be analyzed. With the use of the C2U tool presented in Section 3.4.5, the composite model can be transformed to UPPAAL, and verification can begin.

The analysis of the first layout

Eight automata were created, six of which represent the sections and the remaining two stand for the trains. In order to verify their emergent behavior, queries can be defined which are processed by UPPAAL and evaluated on the network of the automata.

The following requirements are expected to be satisfied at all times:

1. *The system must be deadlock free*
2. *Two separate trains must not be positioned on the same section. If two trains proceed towards each other, both of them have to be stopped on adjacent but separate sections.*

The following queries have been evaluated on the system:

- `A[] not deadlock`, i.e. the system is deadlock free
- `A[] !(Process_main_region_trainOfStatechartOfTrain1.T2 && Process_main_region_trainOfStatechartOfTrain2.T2)`, i.e. train1 is never positioned on section3 completely while train2 is positioned on section4 completely
- `A[] !(Process_main_region_trainOfStatechartOfTrain1.T2 && Process_main_region_trainOfStatechartOfTrain2.T1T2)`, i.e. train1 is never positioned on section3 completely while train2 is positioned on the edge of section4 and section5
- `A[] !(Process_main_region_trainOfStatechartOfTrain1.T1T2 && Process_main_region_trainOfStatechartOfTrain2.T2)`, i.e. train1 is never positioned on the edge of section2 and section3 while train2 is positioned on section4 completely

UPPAAL is able to evaluate these queries on the models and provides answers as the result of an exhaustive state space search. As Figure 5.4 depicts the requirements are satisfied by the designed composition system. This proves the following statements:

1. There is no deadlock in this layout.
2. Two trains proceeding towards each other can not collide, as they are disabled (i.e. their braking starts) right after they reach a section that has only one other section between it and the section occupied by the other train. The braking period in the worst case is the length of a whole section.

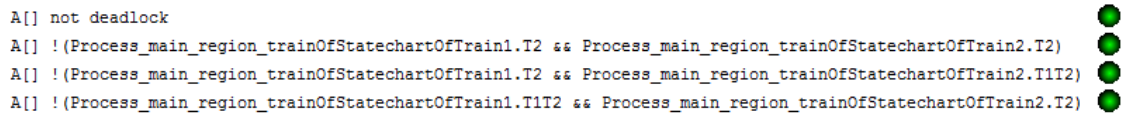

Figure 5.4: The queries that have been evaluated on the system.

The analysis of the second layout

The only difference of this layout from the first one is the orientation of *train2*, i.e. it may proceed towards *section6* instead of *section4*.

The following requirements are expected to be satisfied at all times:

1. *The system must be deadlock free.*
2. *Two separate trains must not be positioned on the same section. If one train proceeds towards another one from the back at least one whole section has to be in between them. If the train in the back breaks this rule, it has to be stopped. The train in the front may keep going.*

The following queries have been evaluated on the system:

- `A[] not deadlock`, i.e. the system is deadlock free
- `A[] !(Process_main_region_trainOfStatechartOfTrain1.T2 && Process_main_region_trainOfStatechartOfTrain2.T1)`, i.e. train1 is never positioned on section3 completely while train2 is facing in the other direction and is positioned on section4 completely
- `A[] !(disabledOfTrain2)`, i.e. train2 can never be stopped by the safety logic in this layout

As Figure 5.5 depicts the requirements are satisfied by the designed composition system. This proves the followings:

1. There is no deadlock in this layout
2. One train proceeds towards another one from the back is stopped (i.e. its braking starts) before it reaches a section that is located next to an occupied section of another train. This does not affect the train in front, it can keep going.

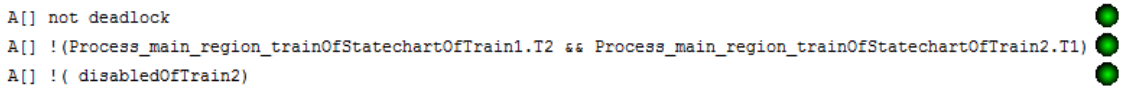

Figure 5.5: The queries that have been evaluated on the system.

In conclusion, the correctness of the safety logic designs has been proven with the help of the framework. As a result, the source code that can be generated from the composition models will also work correctly in these situations owing to the common semantics.

Chapter 6

Conclusion

Model-driven software development is a paradigm that helps to improve the quality of software products by the creation and handling of coherent system models throughout the different phases of development. There are several tools supporting this approach in many ways, e.g. by providing modeling languages, generating source code from models, or providing (formal) verification and validation.

Unfortunately, these aspects are rarely covered in the same tool, but the integration of different formalisms is also cumbersome. This is due to the different semantics and expressiveness of the modeling languages, which requires complex transformations to get from one tool to another.

The framework presented in this work addresses the problem by providing a formal intermediate language for state-based models, reducing the number of necessary conversions to $N + N$ instead of $N \cdot N$. Furthermore, the composition of such models is also supported by the framework, even for heterogeneous formalisms.

By defining a transformation from Yakindu to the intermediate language defined in theta, and from the intermediate language to UPPAAL, the framework is capable of generating the implementation of, and analysis models for a network of statecharts. This capability was demonstrated on the MoDeS3 project, which is a safety-critical demonstrator for distributed systems.

The case-study demonstrated that the framework is indeed capable of generating production-ready software from statechart models, including the verification and validation of the modeled behavior. The validation rules defined for the intermediate language helped in identifying problems early during component modeling. The ability to compose simple models to a complex system greatly reduced the complexity of individual statecharts. The generated UPPAAL automata and queries proved the correctness of the design of the safety logic, the credibility of which is ensured by the common semantics defined in Section 3.3.2.

The presented framework is extensible in several ways. Subject to future work, we plan to add support for the back-annotation of validation and verification results to aid the designer in correcting their mistakes. We also plan to extend the compositional language to allow hierarchical compositions, i.e. the composition of composite systems. The semantics of the composition can also be modified to support different systems, e.g. distributed systems where communication occurs over networks. This would also require the extension of code generators. Since the framework is prepared to be extended with additional

modeling formalisms, a straightforward improvement would be to add support for more engineering and analysis languages.

By building bridges between different tools and formalisms, we hope to support software and system engineers in fully leveraging the potential model-driven software development.

Acknowledgements

This work was partially supported by IncQuery Labs Ltd. and MTA-BME Lendület Research Group on Cyber-Physical Systems.

List of Figures

List of Tables

Bibliography

[1] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[2] Jung Ho Bae and Heung Seok Chae. Systematic approach for constructing an understandable state machine from a contract-based specification: controlled experiments. *Software & Systems Modeling*, pages 1–33, 2014.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Möller, Paul Pettersson, and Wang Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[5] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. pages 87–124. Springer-Verlag, 2004.

[6] Patricia Bouyer and François Laroussinie. *Model Checking Timed Automata*, pages 111–140. ISTE, 2010.

[7] Anne-Marie Déplanche and Sébastien Faucou. *Architecture Description Languages: An Introduction to the SAE AADL*, pages 353–383. ISTE, 2010.

[8] Object Management Group. Information technology – Object Management Group Unified Modeling Language (OMG UML) – part 2: Superstructure. Technical Report ISO/IEC 19505-2:2012, Object Management Group, 2012.

[9] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal methods and testing. chapter Testing Real-time Systems Using UPPAAL, pages 77–117. Springer-Verlag, Berlin, Heidelberg, 2008.

[10] Benedek Horváth. Elosztott biztonságkritikus rendszerek xtUML alapú modellvezérelt fejlesztése. Bachelor's thesis, Budapest University of Technology and Economics, 2014.

[11] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[12] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Trans. Softw. Eng.*, 39(6):869–891, June 2013.

[13] Stephan Merz. *An Introduction to Model Checking*, pages 77–110. ISTE, 2010.

[14] Technical Operations International Council on Systems Engineering INCOSE. INCOSE Systems Engineering Vision 2020. Technical report.

[15] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 2016.