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

Model-Driven Development of Reactive Systems with Mixed Synchronous and Asynchronous Hierarchical Composition

Scientific Students' Association Report

Author:

Bence Graics

Advisor:

Vince Molnár

2017

Contents

Abstract **i**

1 Introduction **1**

1.1 Project Timeline . . . 2

1.2 Overview . . . 3

2 Background **4**

2.1 Modeling . . . 4

2.1.1 Model-Driven Software Development . . . 4

2.1.2 Modeling Languages . . . 5

2.2 State Machine Formalism . . . 5

2.3 Composite Reactive Modeling . . . 6

2.4 Related Work . . . 8

2.4.1 Ptolemy II . . . 8

2.4.2 BIP . . . 9

2.4.3 MATLAB/Simulink Stateflow . . . 10

3 The Gamma Framework **12**

3.1 Overview . . . 12

3.2 The Gamma Statechart Language . . . 12

3.2.1 Constraint, Statechart and Interface Language . . . 12

3.3 Integrating Engineering Models . . . 17

3.4 Validation . . . 18

3.5 Code generation . . . 18

3.6 Verification . . . 19

4 Language Extensions for Composition **20**

4.1 The Composition Language . . . 20

4.1.1 Endpoint Elements . . . 20

4.1.2 Communication Elements . . . 22

4.1.3 Composition Elements . . . 23

4.1.4 Summary . . . 30
4.2 Composition Semantics . . . 30
4.2.1 Synchronous Components . . . 30
4.2.2 Asynchronous Components . . . 33
4.3 Crossroads Example . . . 35
4.4 Extending the Code Generation Functionality . . . 38
4.4.1 Interfaces . . . 38
4.4.2 Components . . . 40

5 Implementation **44**
5.1 Technologies . . . 44
5.1.1 Eclipse Environment . . . 44
5.1.2 Xtext Framework . . . 45
5.2 Architecture . . . 45
5.3 Integrated Modeling Languages . . . 45
5.3.1 Integrated Engineering Language: Yakindu . . . 46
5.3.2 Integrated Model Checker: UPPAAL . . . 47

6 Case Study: MoDeS^3 **50**
6.1 Introduction . . . 50
6.2 Interlocking Safety Logic . . . 50
6.2.1 Interfaces . . . 51
6.2.2 Section Statechart . . . 52
6.2.3 Turnout Statechart . . . 53
6.3 Supporting the Development and Verification . . . 54
6.4 Formal Verification of the Safety Logic . . . 54
6.4.1 Analysis of the First Layout . . . 57
6.4.2 Analysis of the Second Layout . . . 57
6.5 Summary . . . 58

7 Conclusion **59**

Acknowledgements **60**

Bibliography **63**

Abstract

As a result of the recent technological advancements in computation, programmable controllers are now used extensively even in critical domains such as automotive embedded systems. Moreover, in the era of "intelligent" devices, programs are not centralized anymore – for example, the embedded controller directly actuating the vehicle is in close relation with the electronic control unit, which is in turn communicating with services in the cloud. The complexity of such heterogeneous systems may be very high. Model-driven development is widely used to handle the complexity because it supports the developer in focusing on the logical aspects of the problem instead of the technical details.

In this work, we present a modeling tool to answer the following challenges inherent in the systems characterized above.

1. Component-based architecture: the targeted systems are typically composed of smaller components – therefore a suitable modeling language shall support hierarchical modeling.

2. Communication: components usually communicate by means of logical signals or messages – communication shall happen through well-defined ports and interfaces.

3. Distributed components: components often do not constitute a single program, but several pieces of software that run on different pieces of hardware – the resulting heterogeneous communication requires different compositional semantics.

4. Quality and correctness: often, these systems (or parts of them) perform critical tasks where correct operation is fundamental – therefore, their design must be sound and correct, which can be supported by validation and verification, while the quality of the implementation can be ensured with automatic code generation and testing.

In this work we propose the gamma framework, which is a modeling tool to build hierarchical, component-based, reactive systems. Elementary components can be defined in the built-in formal modeling language as well as in third-party tools integrated with gamma (e.g., Yakindu Statechart Tools). The framework supports three types of semantics for composition: asynchronous-reactive semantics for the proper abstraction of distributed communication, synchronous-reactive for components of a single program or for highly synchronous communication, and cascade for the logical decomposition of a single function. The modeling process is supported by live validation both on the component and system level. Model checkers (such as UPPAAL), integrated into the framework and hidden from the user, can be used to ensure the correctness of models. The implementation of the design can be automatically generated from the models, where the quality of the generated code is validated by a set of automatically generated tests.

The extensive functionality and the possibilities provided by the gamma framework are also demonstrated through a railway-themed case study.

Chapter 1

Introduction

Programmable controllers are getting more and more widespread in several industrial fields, such as automotive, railway and avionics domains. Such systems are generally not centralized; they are composed of heterogeneous components that are distributed among several computing nodes. Therefore, the components communicate and coordinate the actions of each other through some communication network. They are often also capable of obtaining information and reaching computing resources (e.g., cloud computing) via the Internet.

As the complexity of such systems increases, new methodologies and tools are required to supervise the design, implementation and analysis of interacting components. The model-driven development approach has been adopted to simplify the development process, aiming to focus on the abstract representations of the activities and knowledge that are important in a certain domain, rather than low-level computing concepts.

Programmable controllers are often considered as reactive systems. State-based formalisms are well-suited for the design of reactive systems, although the resulting models can be complicated and hard to maintain. This problem can be simplified by composition techniques, i.e., hierarchical models are built using smaller and less complex components. The interaction between the components has to be supported by well-defined interfaces and ports. Furthermore, components of a single system often perform various tasks, which might require different compositional semantics. As such systems (or some parts of them) might execute critical tasks, it is important to ensure the correctness of their design. This can be supported with verification and validation techniques, whereas correctness of the implementation of the system can be ensured with automatic source code generation.

This work introduces the gamma framework, a modeling tool that aims to support the aforementioned needs, i.e., the development of component-based, hierarchical reactive systems. The framework is open to integration with modeling tools (currently Yakindu Statechart Tools) supporting the definition of state-based models, which can be compiled into gamma statecharts. Based on the gamma statechart formalism, the formal composition language of gamma can be used to define hierarchical composite systems. A composite system can be defined according to one of the following semantics.

- **Asynchronous-reactive:** Such models represent components that are executed independently. Asynchronous components communicate with each other by means of messages and message queues. This semantics is convenient for the design of communicating microservices running in different processes.
- **Synchronous-reactive:** A synchronous model represents a coherent unit with a single functionality consisting of concurrently (but not independently) running components.

Contained components communicate in a synchronous manner using signals. This semantics is suitable for the design of HW-dependent, programmable controllers.

- **Cascade:** Cascade components are special synchronous components where the execution of the contained components is sequential: every component receives the output of the previous ones. This model is beneficial for the design of units with a pipeline-like execution: each contained component can be considered as a filter that processes incoming signals and forwards them to subsequent filter units. Therefore, this model supports the design of adapters, runtime monitors and units with a batch-like execution.

The design of both elementary gamma statecharts and composite systems is supported by validation rules that provide feedback to the user about their models at design time. The implementation process is facilitated by automatic code generation. The generated source code conforms to the formal compositional semantics. Formal verification of composite models is also supported by the integration of the UPPAAL model checker. The whole model checking process is hidden from the user with a GUI that facilitates the construction of formal requirement queries.

One of the main priorities of the gamma project is to provide a freely available and extensible framework with model-based technologies to support the research and development of complex reactive systems. The architecture of the gamma framework has been designed considering extensibility, so that researchers can easily integrate their work with the existing components of the framework. Such extensions may include both engineering and formal modeling tools.

1.1 Project Timeline

This section briefly summarizes the evolution and the planned future of the gamma project to put this work in context.

Immediate antecedents The initial research and development goal leading to the design of the gamma framework was the desire to formally verify statechart models built in the open-source Yakindu Statecharts Tools. To achieve this, a two-step model transformation to UPPAAL has been implemented using the semi-formal statechart representation of another project of the research group as the intermediate representation. This transformation is still the part of the framework core.

The first prototype of the Gamma framework The semantical inconsistencies of Yakindu and the need for integrating code from multiple statechart models (in the MoDeS^3 project[1]) led to the design of the first version of the gamma framework. The goal was to enable the composition of communicating statecharts to build composite systems. A heavy emphasis of the initial research goal – i.e., formal verification of the models – led to a synchronous compositional semantics that is restrictive enough to make model checking feasible, while still enabling useful communication patterns between the components. The design and implementation of the framework was presented in a Scientific Students' Association Report in 2016 [16].

Formalization of the synchronous compositional semantics The previously implicitly defined semantics of the composition (defined by means of model transformations) has been formally defined in a conference paper [18].

[1] https://inf.mit.bme.hu/research/projects/modes3

Hierarchical composition and ports – Gamma 1.0 The next phase included the introduction of hierarchical composition, i.e., composite systems could be used as components of another composite system. Along with this improvement, the concept of ports and interfaces were introduced to define the "signature" of components and couple related events into well-defined points of service. A part of this work has been presented in the Bachelor's Thesis of the author [17], also including the back-annotation of the verification results both in a textual format and as tests for the generated code (i.e., witness behaviors returned by the model checker were transformed to use the concepts of the original model, and were used to generate tests). The port system and improvements of the verification process is presented in this work for the first time. A brief description of previous work is presented in Chapter 3. This version of the gamma framework is publicly available along with a tutorial.[2]

Gamma 2.0 In the *current phase of the project*, our goal is to broaden the modeling power of composition by introducing the cascade and asynchronous composition semantics. We are now focusing more heavily on functionality and expressive power, replying to the various feedbacks given to the first version. This work addresses the main challenge: the design of the syntax and precise semantics of the extended composition language. This will serve as the pivot for a full-fledged implementation of the code generator for the new compositional modes as well as for their transformation to the formal language of UPPAAL.

Built-in code generation An ongoing work supporting the current version of the gamma framework is the design and implementation of a built-in code generator for statechart models. Currently, the framework relies on external code generators. Completing this project will enable the direct usage of the statechart formalism of gamma.

Side projects There are many side projects building on the gamma framework. They include code generation to distributed controllers (with network communication), a simplified, but rigorously validated statechart formalism, and an extension that enables the specification of contracts for the ports by means of sequence charts (with validation and runtime monitoring).

Gamma 3.0 Once the semantical improvements are finalized and included in the code generation and verification functionalities, we plan to improve the already extensible architecture of the gamma framework and introduce new modeling formalisms, verification tools, code generators (e.g., to C/C++) and potential model reductions.

1.2 Overview

The rest of the work is structured as follows. Chapter 2 presents the theoretical concepts behind the gamma framework and three related modeling tools. Chapter 3 describes the state of the gamma frameworkfrom where the current work started. Our main contribution, i.e., the extension of the gamma language and the code generator, is introduced in Chapter 4. Chapter 5 presents the architecture and the employed technologies of the gamma framework as well as the integration of third-party modeling languages. The applicability of the gamma framework is demonstrated through a railway-themed case study in Chapter 6. Finally, Chapter 7 provides concluding remarks and plans for future work.

[2] http://gamma.inf.mit.bme.hu/

Chapter 2

Background

This chapter presents the notions and ideas necessary to understand the rest of the work. As a motivation of the gamma framework, we start with the introduction of the model-driven software development paradigm, which is the approach in which the framework has been conceived. Then, we describe the state machine formalism that we use to represent models. Next, we introduce the concept of composite reactive modeling, which is the basis of this work. Finally, we present existing composition modeling solutions related to the gamma framework.

2.1 Modeling

Model is a primary concept in several fields of study. Generally, in software and system engineering the term model is used in the following sense: a *model* is a simplified image of an element of the real or a hypothetical world (the system), that replaces the system in certain considerations. A model is always based on an *original subject* (the system) highlighting some of its features while neglecting some others. This way the model becomes competent to be used in the place of the original element with respect to a certain purpose [10].

Models can be either *structural* or *behavioral*. Structural models (class diagram, component diagram, etc.) emphasize structural aspects of the system with respect to managed data or to architecture. On the other hand, behavioral models (activity diagrams, statecharts, etc.) focus on the dynamic behavior of the system by describing how they are executed.

2.1.1 Model-Driven Software Development

Model-driven software development (MDSD) is a software development methodology that uses models as the primary artifact and main information source in each phase of the development process [33]. By putting models in focus, the MDSD approach aims to *1)* enhance productivity via recommendations and best practices in the application domain, *2)* simplify the design process by using patterns and early validation with modeling tools and *3)* maximize compatibility and ease communication between teams and individuals working together by standardizing terminology and using both general purpose and domain-specific languages. As a consequence, the MDSD approach should reduce the cost of development and increase the quality of the designed software [10].

Defining it this way, MDSD is a rather general concept aiming to put models into the focus in the software development process while heavily relying on modeling technologies. There are several subsets of MDSD giving more concrete guidelines on the development of software sys-

tems recommending modeling techniques and technologies, such as *model-driven architecture* [26] and *model-centric software development* [35].

2.1.2 Modeling Languages

Creating precise, interpretable models requires an environment that defines the rules of model creation. This environment is provided by modeling languages.

Definition 1 (Modeling language). A modeling language consists of the following elements:

- Metamodel: a model defining the building blocks of the modeling language as well as their relationships.
- Concrete syntax: a set of rules defining a graphical or textual notation for the element and connection types defined in the metamodel.[1]
- Well-formedness constraints: a set of constraints that models have to meet in order to be deemed valid in the modeling language.
- Semantics: a set of rules that define the meaning of the element and connection types defined in the metamodel. Semantics can be classified as follows:
  - Operational: operational semantics defines what should happen during execution.
  - Denotational: denotational semantics is given by translating concepts in a modeling language to another modeling language with a well-defined semantics. Thus, the meaning of the modeling elements are implicitly given.

Regarding the portrayal of models, modeling languages can be graphical (UML, Ptolemy II) or textual (C#, Verilog). There are several modeling languages (Yakindu, BIP) that employ graphical and textual notations side by side, exploiting both of their advantages.

As for application domains, modeling languages can be partitioned into *domain-specific* (AADL, Autosar, MATLAB Stateflow) and *general purpose* modeling languages (UML, SysML, Petri nets). Domain-specific modeling languages are tailored to a certain application domain, e.g., avionics, automotive, business modeling, whereas general purpose modeling languages are broadly adaptable across application domains and lack specialized features for a particular domain. The line is not always sharp between the two, as a modeling language might have specialized features for a certain domain but can also be applicable in other fields.

2.2 State Machine Formalism

The main functionality of the gamma framework presented in this work is supporting the composition of separately defined statecharts. This section briefly introduces the theory of *state machines* – a formalism on which the widely-used statechart formalism is based.

State machine is a mathematical model of computation to describe the behavior of a system, component or object in an event-driven way [3]. Formally, a deterministic, fully specified finite state machine is a 5-tuple: $M = (S, s^0, I, O, T)$ where:

- $S = \{s_1, s_2, \cdots, s_n\}$ is a finite set of states, i.e., stable situations of the state machine. $s^0 \in S$ is the initial state.

[1] A single modeling language can have multiple syntaxes.

- I is a finite set of input events that are stimuli from the environment and O is a finite set of output events that are stimuli for the environment such that $I \cap O = \emptyset$.
- T: $(E \times S) \rightarrow (S \times O)$ is the fully defined transition function that represents changes of states in response to input events and generating output events meanwhile.

There are various extensions to the state machine formalism that facilitate the compact modeling of hierarchical and concurrent systems [23]. The most relevant one to this work is statecharts [22], which also supports auxiliary variables in addition to supporting concurrency and state refinement. Statecharts are generally represented graphically, although there are modeling languages, such as gamma, that support their textual description. The graphical representation of the most important elements of statecharts are depicted by Figure 2.1.

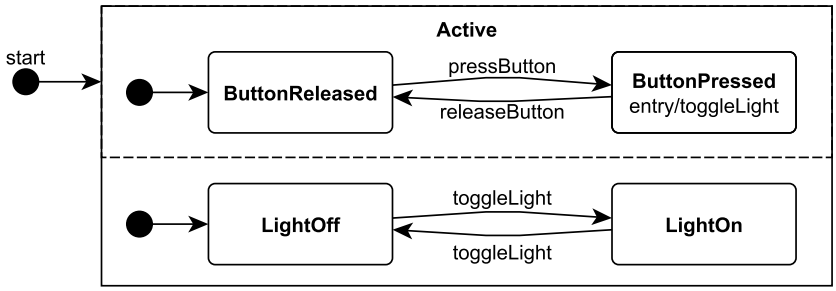


Figure 2.1: The graphical representation of the most important elements of statecharts.

Regions contain a set of states which are situated on a single hierarchy level. Regions are represented by coherent areas, whereas states are represented by rounded rectangles. Composite states can contain one or more regions that contain additional states. Each region has a single entry node, which is represented as a black circle. The initial state of a particular region is denoted by the transition coming out of the entry node. Transitions are represented as arrows. Events associated to transitions are represented with their names.

2.3 Composite Reactive Modeling

Beyond a certain complexity, systems cannot be designed without composition techniques. However, for a well-defined system behavior, the semantics of composition needs to be defined precisely by means of a *model of computation*. A model of computation is a set of rules defining *1)* what constitutes a system component, *2)* the government of concurrent execution of system components and *3)* the communication between system components [11]. A *semantic domain* is the implementation of a certain model of computation [11].

The rest of the section introduces some models of computation related to the gamma framework. Note that some of these models might have potential variants, the descriptions given here cannot be considered universal.

Dataflow In the dataflow model a system component communicates with its environment via input and output message queues. Message queues contain tokens that can be considered as messages of certain types. The behavior of a single component comprises of a sequence of firings. A single firing is initiated in response to a given combination of available input tokens,

i.e., the arrival of particular data serves as trigger for a component. A firing consumes the corresponding input tokens and produces a defined combination of output tokens.

An advantage of the dataflow domain is that it provides opportunity for statical analysis of deadlock-freedom and boundedness. Schedulings of components can also be computed statically [14]. Dataflow models are convenient for the representation of streaming systems, where sequences of data flow in definable patterns between system components. For example, signal processing systems, including video and audio systems are especially good application domains. The execution semantics is usually defined in terms of Petri nets [32].

Asynchronous-Reactive In the asynchronous-reactive model, system components represent concurrent processes that communicate with each other using message queues [27]. Writing to the message queues always succeeds instantly, whereas reading from an empty queue blocks the reader process. This nonblocking-write, blocking-read approach guarantees the determinism of the model [24]. Message delivery is assumed to be dependable, therefore, the sender does not receive nor expect any confirmation (send and forget approach). Messages arrive into the target message queue in the same order they are sent. A single read operation always retrieves a single message from the queue. Additionally, prioritized queues can be introduced to reorder the incoming messages and prefer the urgent ones in the read operation.

The asynchronous-reactive model describes concurrent processes whose executions are not depending on external triggers, but are constantly running. Therefore, there is no guarantee for the execution time and execution frequency of system components. It can be considered as a generalization of the dataflow model where system components are concurrently executing processes rather than components reacting to certain incoming token combinations [29].

Synchronous-Reactive The synchronous-reactive model has a notion of time and follows the semantics of synchronous programming languages [8, 19, 13]. In this model system components communicate with each other using signals which are transmitted and received through ports. Furthermore, the execution is driven by a *clock* which emits *ticks* (clock signals). System components are executed in response to the clock signals. Upon execution, a component reads the signals from its incoming ports and transmits signals through its outgoing ones. Generally, components can be considered as functions mapping values from their incoming ports to their outgoing ports depending on their current state. The output signals are sustained until the next tick. The input signals are sampled only upon ticks, changes of signals in between ticks are ignored. Contrary to dataflow and synchronous-reactive models, signals can be *absent* and components are also able to react to such cases and even to a combination of signals. Since in this model ticks serve as triggers of execution, a combination of signals can be considered as guard expressions of certain activities of components.

The synchronous-reactive model is convenient in situations with complicated control flow, where a system component might take different actions depending on whether a signal/message is present or not. This model is able to handle these situations without possibly nondeterministic communication using synchronization instead. Therefore, it is excellent for the modeling of logical circuits. On the other hand, the synchronous-reactive model is considered "less concurrent" than dataflow or asynchronous-reactive models, as the components are executed in a lockstep fashion for every clock signal. Consequently, such models are better to describe a single-threaded component in a logically decomposed way.

2.4 Related Work

There are several software tools that aid the process of system design by supporting the composition of components. This section introduces three frameworks that had influence on the design of the gamma framework: Ptolemy II, BIP and MATLAB/Simulink Stateflow.

2.4.1 Ptolemy II

Ptolemy II[2] [14, 11] is an open-source modeling framework that supports the modeling and simulation of hierarchical composite systems with various component implementations and interaction semantics. It is being developed by researchers at UC Berkeley. Ptolemy II is particularly strong at supporting simulations with component-based designs of diverse semantics. Components, which are called *actors* in this framework, can be regarded as concurrently executable software modules. Furthermore, they are able to interact with each other by sending messages through interconnected *ports*. Ptolemy II models are created by the composition of actors, which is supported on multiple hierarchy levels.

Actors are independent software components whose interactions can be executed with different *semantic variations*, defined by *models of computation* (MoC). Ptolemy II offers numerous MoCs that rigorously define the interaction between actors.

- In *process networks* (PN) actors represent concurrent processes, communicating by sending messages to the message queues of each other.
- The *rendezvous* domain is similar to PNs, but communication is achieved via synchronization (also known as handshake). When a process arrives to a synchronization point, it has to wait for the other participant to be able to synchronize.
- *Synchronous dataflow* (SDF) is a type of dataflow model where actors are executed when their required data inputs (tokens) become available. They consume a fixed amount of tokens from particular input ports and produce a fixed amount of tokens to the corresponding output ports.
- In the *synchronous/reactive* (SR) domain execution follows *ticks* of a global *clock*. The actor implements a stateful function that maps the values of its input ports to the values of its output ports at each tick.
- In the *discrete events* (DE) domain actors use *events* for communications, which are placed on a time line. Each event has a time stamp and a value. Actors process events in chronological order.
- The *continuous-time* domain models ordinary differential equations. Every connection in this domain represents a continuous-time function.

The implementation of a MoC is called *director* in Ptolemy II. Each level of hierarchy in a model must have a single director that specifies the MoC. However, directors of various hierarchy levels may have different types. Still, the composition of such heterogeneous components adheres to a rigorous semantics which is a very powerful facility of Ptolemy II. Furthermore, Ptolemy II offers additional compositional possibilities: actors can be combined with finite state machines, which results in *modal models* [28]. In these cases each actor is associated to a single state serving as its refinement. This way, the prevalent actor implementation inside a component can dynamically change during runtime where the possible changes are defined

[2] http://ptolemy.eecs.berkeley.edu/

by the particular state machine (with its active state) and the associated actors. Additional notable composition modes are as follows: the (hierarchical) composition of state machines and continuous-time models results in a hybrid system [31]; by composing the SR models with state machines we get StateCharts [30]. Ptolemy II supports the simulation of models.

The idea of mixing well-defined MoCs that control the interaction of components heavily influenced the gamma framework that aims to support synchronous and asynchronous reactive models. Compared to Ptolemy II, the gamma framework offers source code generation functionalities that is not supported by the latest Ptolemy II versions[3]. What is more, the gamma framework supports the formal verification of certain composite models, which is not available in Ptolemy II.

2.4.2 BIP

BIP[4] [9, 2, 4] (Behavior, Interaction, Priority) is a general purpose modeling framework that supports the formal definition of heterogeneous systems. It is being developed by the VERIMAG academic research laboratory. For the definition of system models BIP offers a design flow[5] that aims to ensure maximum consistency between the consecutive design steps. The design flow is supported by the BIP language as well as a comprehensive tool set [6, 15].

The BIP language supports the layered definition of hierarchical composite systems [37], defining three layers. The lowest layer specifies the behavior of system components, atomic or compound, defined using a variant of the Petri net formalism extended with variables and functions described in C. *Ports* are also defined in this layer. The intermediate layer consists of a set of connectors linking ports together, thus defining the interactions between transitions of contained components. The top layer includes a set of dynamic priority rules between interactions and can be used for the specification of scheduling policies. This layered architecture provides a clean separation of internal behavior (lowest layer) and communication (intermediate and top layer) in compound components.

BIP defines a clear operational semantics that describes the behavior for both atomic and compound components. The behavior of atomic components are based on a rigorous transition system model, whereas the behavior of compound components is described as a composition of the behaviors of its (atomic or compound) components [7].

The BIP design flow is backed up by a comprehensive tool set that implements numerous functionalities. The tool set contains several transformers supporting the conversion of third-party models (MATLAB/Simulink, AADL, C, GeNoM) to BIP models, which is called language embedding by the developers. Furthermore, there are transformers aiming to transform various BIP models into other BIP models, such as *system models* to *distributed system models*. The tool set also contains code generators that are able to produce C/C++ or Java source code from BIP models. Finally, formal verification of invariant properties and deadlock-freedom is also supported. Recently, an online design studio has been released for BIP which supports the graphical edition of BIP models as well as simulation and consistency check.[6]

BIP influenced the gamma framework in several aspects, e.g., supporting multiple design languages, hierarchical composition, the generation of source code and formal verification capabilities. The most significant difference is that the gamma framework puts the focus on statecharts, instead of Petri nets and C-like functions, which we believe to be a more engineer-friendly modeling formalism.

[3] http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIfaq.htm#CodeGen
[4] http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en
[5] http://www-verimag.imag.fr/The-BIP-Design-Flow.html?lang=en
[6] The design studio is available at `https://editor.webgme.org/`.

2.4.3 MATLAB/Simulink Stateflow

Stateflow[7] [25] is a commercial framework for the modeling and simulation of reactive systems. It can be considered as the de-facto standard for compositional state-based modeling in the domain of safety critical embedded system design [12]. In addition to supporting the logical design of single components, Stateflow can integrate components using various scheduling algorithms, simulate composite models, validate system models and generate source code.

Stateflow supports the definition of component behavior with two formalisms: statecharts and flowcharts. The statechart language is a variant of the hierarchical finite state machine notation introduced by Harel [22], supporting hierarchical representation, orthogonal regions, history states and state activities. On the other hand, the flowchart language, which supports the description of *functions*, has more unique characteristics. Flowcharts describe internal behavior of states. A flowchart consists of junctions that are connected with transitions. Transitions can contain guards and actions. Actions can contain MATLAB and Simulink functions or even custom C code. Contrary to states, a junction is left immediately after it is entered. A single execution of a flowchart lasts until a terminal junction (a junction without outgoing transitions) is reached or all guards of the outgoing transitions of the junction are evaluated to *false*. The execution of flowcharts, if their containing states are active, can be initiated by certain triggers, such as clock ticks or entry/exit events.

One of the main merits of the Stateflow framework is that it supports the combination of statechart diagrams with flowchart diagrams in a versatile way. A single execution of the Stateflow model, called a step, is as follows:

1. Determine the active state.
2. Examine whether an outgoing transition of the state is enabled. If so, fire it and the step ends.
3. Otherwise execute internal actions (flowcharts, then *during actions*).
4. Repeat this process recursively for any internal state that is active.

Formalizations of the Stateflow operational semantics can be found in [21, 36], whereas [20] introduces a denotational semantics.

After creating a statechart model, it can be converted into an atomic component, called *subchart*, and reused in composite models. Additionally, an interface has to be defined for the subchart that specifies the events that can be sent to or received from the adjacent subcharts. In addition to subcharts, composite models consist of links that are responsible for the propagation of events between subcharts and a scheduling algorithm that specifies the execution order of components. Components are activated on event reception – they process the event as described above, potentially generating other events and activating other components. The scheduling algorithm can be defined using conditional and time-based logic, which support event-based and time-based operators (*every*, *at*, *before* and *after*). These operators can be used for the measurement of time and the counting of event occurrences. Thus, the need for separate timer and counter components are eliminated [1].

Stateflow provides additional functionalities that are very useful for system design, system validation, verification and implementation. Composite system models can be simulated, which involves highlighting active states, firing transitions and displaying the values of variables [34]. The Simulink Verification and Validation add-on supports traceability during development, which includes the linking of Stateflow objects to requirement specifications defined in IBM

[7] https://www.mathworks.com/products/stateflow.html

Rational DOORS. Additionally, this add-on supports the static analysis of Stateflow models, checking whether the particular model is compliant to certain standards. Furthermore, design errors can be detected using the Design Verifier tool, which builds on formal methods. Finally, HDL, PLC or C/C++ source code can be generated directly from the composite models using add-on code generator products.

The concept of composing statecharts is one of the basic functionalities of the gamma framework, therefore we consider the work of Stateflow very important as the de-facto standard of the domain. On the other hand, Stateflow is a commercial product, which makes it difficult to study and extend it, which is imperative in research context. Moreover, the gamma framework aims to support multiple modeling statechart-like languages as front-ends, realizing the interactions of heterogeneous models. Contrary to our goal, Stateflow – as most commercial tools do – strives to be self-contained, limiting the extensibility of the modeling formalisms.

Chapter 3

The Gamma Framework

The gamma framework introduced in this work is designed to support the model-driven development of safety-critical systems. It is based on the framework presented in the BSc thesis of the author [17], but since then, it has been extended with numerous fixes and improvements addressing both functionality and user-friendliness. In this section we briefly introduce the gamma language, consisting of multiple parts, as well as additional functionalities of the gamma framework since the results and solutions of this work heavily rely on them.

3.1 Overview

The goal of the gamma framework is to support the model-based design, implementation, validation and verification of reactive systems (see Figure 3.1). All functionalities rely on the gamma statechart language (introduced in Section 3.2), which provides a common basis regarding model elements and semantics. Statecharts created with the gamma statechart language can be composed using the gamma composition language (introduced in Section 4.1). The gamma framework supports system design by the integration of high-level engineering modeling languages as software designers like to work on a high-abstraction level (Section 3.3). Validation is supported by well-formedness constraints to give feedback to designers on the quality of their models at runtime (Section 3.4). Furthermore, implementation is facilitated by automatic source code generation from composition models (Section 3.5). Verification is supported by the integration of formal modeling languages, thus enabling model checking facilities. The framework also supports the generation of test cases (Section 3.6).

3.2 The Gamma Statechart Language

The huge abstraction gap that separates the design and formal modeling domains, both supported by the gamma framework, needs to be bridged by an intermediate language. For this purpose we designed the gamma statechart language. Generally, this language provides a common basis for all functionalities of the gamma framework. Furthermore, the integration of additional engineering and formal languages is facilitated by this language.

3.2.1 Constraint, Statechart and Interface Language

The gamma language has four distinct parts: *constraint*, *statechart*, *interface* and *composition*. Each part is in close relation with the others and have certain classes that serve as interaction

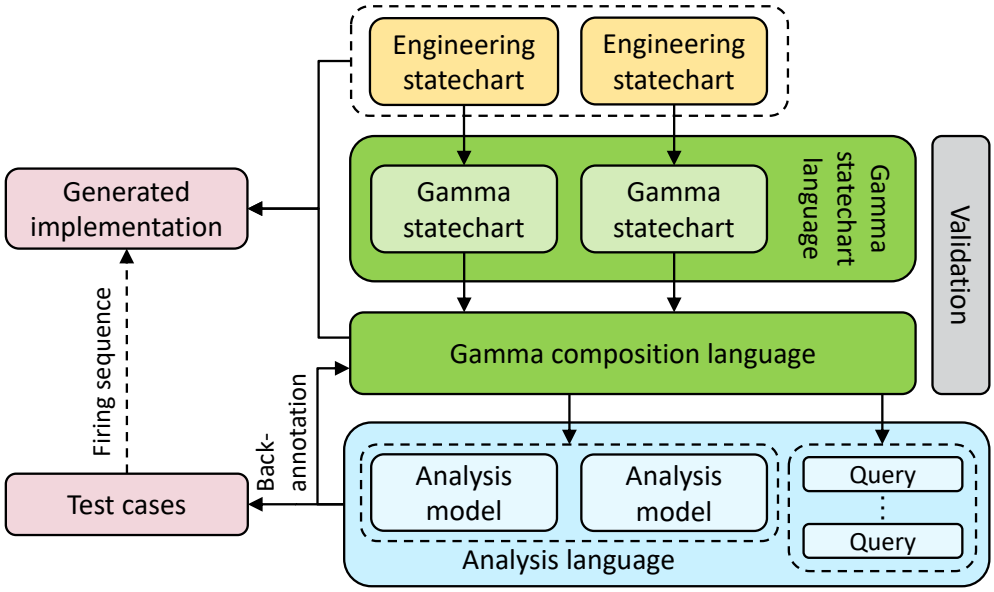


Figure 3.1: The functionalities of the gamma framework.

points between them.[1] Figure 3.2 depicts the dependencies between the parts of the gamma language. The following sections briefly introduce the constraint, statechart and interface languages of gamma. As the main contribution of this work is the design of the composition language it is presented separately in Chapter 4.

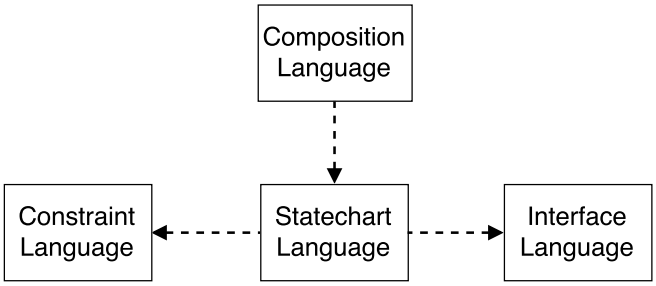


Figure 3.2: Dependencies between the parts of the gamma language.

Constraint

The **constraint** part is the basis of the metamodel that does not depend on any other parts. It supports the definition of *constraints*, which is a general concept for type definitions, variable, structure, function declarations and the specification of expressions. *Integer*, *natural*, *boolean*, *real* and *enumeration* types are supported by the language. Furthermore, there are more than forty classes facilitating the specification of expressions including arithmetic, logical and assignment expressions. The functionality of the constraint part of the metamodel is needed to declare and handle variables in statecharts. Table 3.1 summarizes the supported operators of the constraint language.

[1]Such classes are going to be marked with a dark gray background in class diagrams in the following subsections.

Table 3.1: Brief description of the supported operators of the constraint language

Prec.	Name	Operator	Description	Grouping
1	Implication	imply	implication	Right-to-left
2	Disjunction	or	logical OR	N-ary
3	Conjunction	and	logical AND	N-ary
4	Negation	not	logical NOT	Right-to-left
5	Equality	=, /=	equality/inequality	Right-to-left
6	Relational	<, >, <=, >=	comparison operators	Left-to-right
7	Addition	+, -	addition, subtraction	Left-to-right
8	Scaling	*, /	multiplication, division	Left-to-right
9	Modulo	mod	modulo	Left-to-right
10	Euclidian division	div	Euclidean division	Left-to-right
11	Sign	+, -	unary plus, unary minus	Left-to-right

Interface

The **interface** part supports the definition of interfaces, which serve as contracts between interacting components of gamma models. These contracts apply to the ability of *dispatching* and *receiving* certain events. Events represent occurrences of some importance. Figure 3.3 depicts the interface part of the metamodel. *StatechartInterfaces* is the root element, and is responsible for containing interfaces. Interfaces contain event declarations, which contain the corresponding events, and also specify their directions. Directions can be *in*, *out* or *in-out*; the latter represents events that can be used as both *in* and *out* events. Events can contain parameter declarations, which provide additional information about the corresponding event. Furthermore, an *inheritance* relationship is defined between interfaces: an interface inheriting from other interfaces contains each event declaration of its parents and is permitted to contain additional ones. This way, a particular interface can be used in each place where its ancestor is expected. An example interface definition can be seen below.

```
interface Base {
    in event baseEvent(param : integer)
}

interface Descendant extends Base {
    inout event descendantEvent
}
```

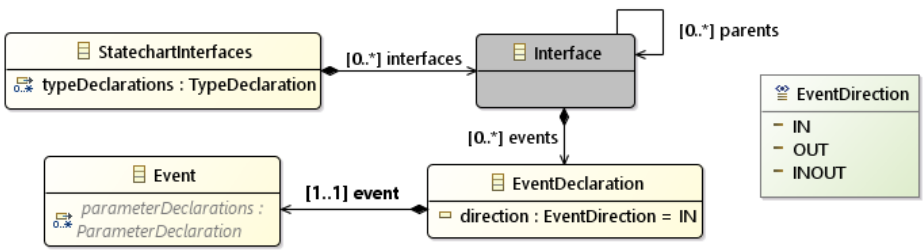


Figure 3.3: The interface part of the gamma metamodel.

Statechart

The main goal of the gamma statechart language is to support the rigorous design of reactive systems while providing conventional facilities of modern statechart languages. To support strictness, a well-defined semantics is needed. The language is given a denotational semantics described in [17] by mapping gamma model constructions to the elements of a formal timed automaton implementation. Moreover, the most important elements of the language are presented in this section.

The **statechart** language supports the definition of hierarchical state machines with variables. This part of the metamodel contains about thirty classes; Figure 3.4 depicts its most relevant part. The following paragraphs introduce the most important elements of the statechart language.

The syntax of the language is presented with an example statechart *Timer*. This statechart measures time when it is in state *Measuring*. State *Measuring* can be activated with signal *start*; the measurement can be stopped with signal *stop*. The elapsed time is stored in integer variable *elapsedTime*.

```
statechart Timer [
    // Port for communication
    port control : provides Control
] {
    /// Integer variable
    var elapsedTime : integer := -1
    // Transition without trigger
    transition from Initial to Idle
    // Transitions with triggers
    transition from Idle to Measuring
      when control.start / assign elapsedTime := -1
    transition from Measuring to Measuring when control.tick
    transition from Measuring to Idle when control.stop
    // Main region
    region main {
        // Initial state
        initial Initial
        // Simple state
        state Idle
        // Simple state with entry action
        state Measuring {
            entry / assign elapsedTime := elapsedTime + 1
        }
    }
}
```

Statechart definition Statechart definition is the root element of the metamodel. It contains variable declarations, transitions and regions. Regions contained by a statechart definition are considered as top regions; if there are multiple top regions, they are parallel, i.e., they are executed simultaneously.

Variable declaration A variable declaration serves as a symbolic name for a particular *value*, where this associated value can be changed during execution. The supported types of variables are presented in the Constraint part the of metamodel. Furthermore, the variables can be given an initial value using an *expression*.

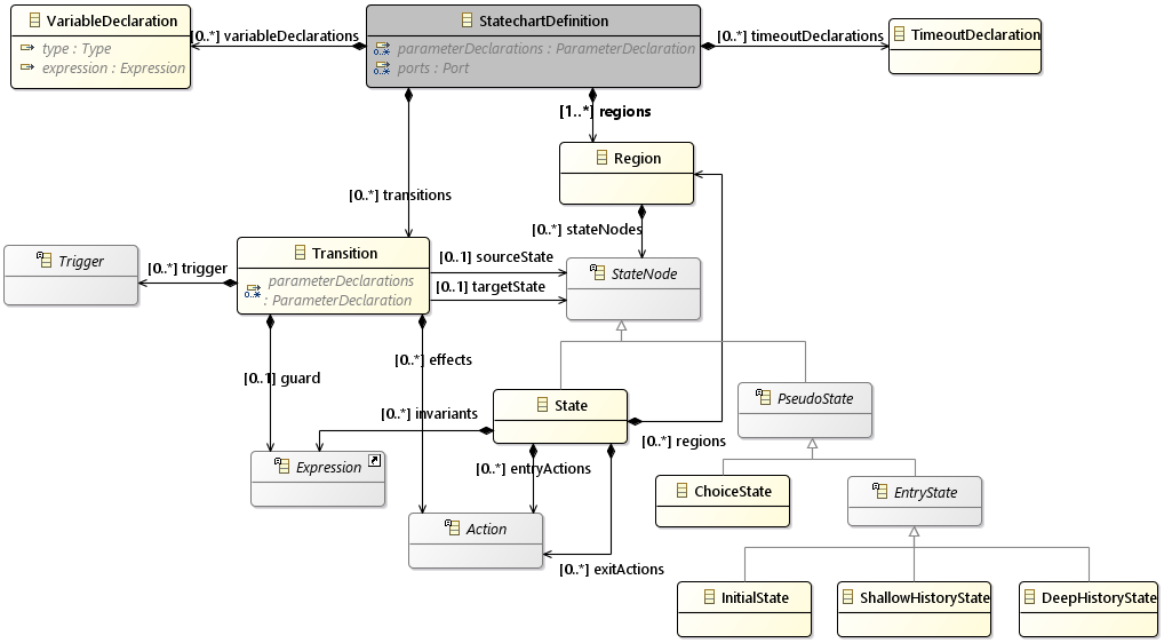


Figure 3.4: The part of the gamma metamodel supporting the definition of statecharts.

Timeout declaration A timeout declaration can be considered as a timer. Timeout declarations can be set by assigning a time specification to them. When the specified time expires, a timeout event is emitted, which can serve as a trigger for actions.

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. A region must contain a single entry state.

Initial state An initial state is used for specifying the first active state of a region after the region is entered. Only one transition can leave it, the target of which defines the first active state of the particular region. This particular transition must not contain either triggers or guards.

History state A history state can be either *shallow* or *deep*. 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 initial state). Similarly to initial states, the transition must not contain either triggers or guards. 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, *invariants* can be defined for gamma states that specify certain conditions that must hold while the particular state is active; if a particular invariant does not hold, the defined model is considered erroneous. Invariants can be used in

verification processes. *Composite* states extend *simple* states with the ability of containing one ore more regions. If a particular state contains multiple regions, they are parallel.

Transition Transitions specify state changes in a statechart. A transition has a single source and a single target. Additionally, a transition can connect state nodes of different regions, unless these regions are parallel. A transition, if not coming out of an entry state or choice state, must contain a trigger, and can contain a guard and effects (actions). During execution, a transition can fire if *1)* its source state is active, *2)* the event referenced by its trigger is raised, *3)* its guard (if it has one) is evaluated to true and *4)* no transition is active on a higher hierarchy level. Unguarded transitions can fire if the corresponding event is raised. If multiple transitions are active at a time on the same hierarchy level, one of them is chosen nondeterministically for firing. A firing transition executes its assigned actions if it has any. This can be either an update of a variable or the raising of an event.

Choice state Choice states are a syntactic sugar used for splitting transitions. Each time a choice state is entered, all guards of its outgoing transitions are evaluated according to the creation order of the transitions. If a guard is evaluated to true, the corresponding transition fires. Choice states are useful for avoiding "code" duplication (trigger and action specifications).

Triggers Triggers specify events on which certain executions can be initiated, e.g., a transition can be fired. As Figure 3.5 depicts, the gamma language supports the definition of complex triggers. A complex trigger, consisting of simple triggers, can describe the relation of multiple triggers as logical relations. A complex trigger may initiate a particular execution only if the corresponding logical relation is evaluated to true. Note that an execution can be initiated on the absence of a certain event using *unary triggers*. Furthermore, simple triggers can be classified into any triggers and event triggers. An any trigger can match any event that has been raised in a particular execution cycle. An event trigger can refer to one of the following events:

- Clock tick reference: this event indicates that a particular clock has emitted a tick.
- Timeout event reference: this event indicates that a certain timeout has been reached.
- Port event reference: this event indicates the reception of a particular event through a particular port.
- Any port event reference: this event indicates the reception of any kind of event through a particular port.

3.3 Integrating Engineering Models

The integration of engineering models is realized via model transformations which map high-level models to the intermediate language of gamma. This enables designers to use the functionalities of the gamma framework without the manual mapping of their models. Also, this approach can support the interaction of various engineering tools with the gamma language as a base point. For each supported engineering language a separate model transformation needs to be defined. This is encumbered by the large element set and non-precise semantics of such engineering languages. Currently, Yakindu is integrated into the framework as an engineering tool. All transformation rules can be found in [17].

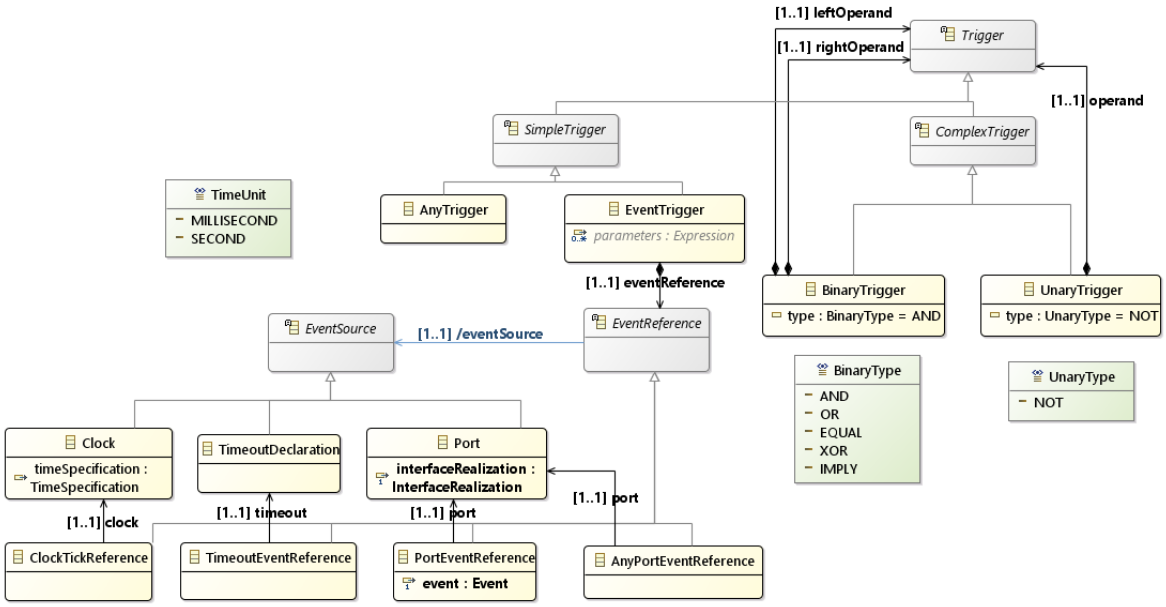


Figure 3.5: The part of the gamma metamodel supporting the definition of triggers.

3.4 Validation

Validation as a static analysis technique takes place on two levels in the gamma framework: on the level of separate statechart models and during the composition of statechart components. The former is realized by thirty formally defined graph patterns that specify ill-formedness constraints. Such constraints include unused variables, race-conditions and enshadowing transitions. If a statechart model under validation violates any constraints, the designer is notified runtime with the display of the incorrect element. All validation rules can be found in [17].

The composition of statechart components is also an error-prone process which can be aided nicely with static analysis techniques. On this level the ill-formedness constraints are also specified as graph patterns, though their implementation mode differs from the one on the level of separate statechart models. Ill-formedness constraints on this level include the connection of ports with invalid types (interfaces) and circular references in the composition hierarchy.

3.5 Code generation

The gamma framework supports automatic source code generation from composite system models. During this process almost all elements specified in the composition phase is utilized. Interface implementations are generated from the interfaces defined in the composition phase. These interfaces are realized by the corresponding port objects of the generated composite system implementations. As a result, users can reach the generated composite systems through familiar interfaces.

Composite system implementations wrap the necessary statechart implementations and subordinated composite systems and construct the required connections (channels) between them. Thus, wrapped components are able to communicate with each other by dispatching and receiving gamma event objects. It is essential that the behavior of composite systems conforms to a rigorous semantics introduced in Section 4.2. Furthermore, composite system implemen-

tations support the registration of observer objects which can be set to detect certain events. These observers are notified on the reception of event objects which they can handle according to their implementations.

3.6 Verification

Formal verification is realized via the integration of *formal modeling languages* capable of model checking. Currently, UPPAAL is integrated into the framework as a model checking tool. The integration is achieved with the use of model transformations mapping composite system models to the formal domain. The transformation rules can be found in [17]. Similarly to source code generation, it is important to ensure the generated models of the formal domain behave according to the semantics of the composite systems. This is achieved with auxiliary models responsible for the conduction of interactions among components.

To utilize the model checking facilities, temporal logic expressions need to be constructed that can be considered as the formalizations of requirements. As the specification of such expressions is cumbersome, their construction is supported by a graphical interface with fillable patterns. Furthermore, to make the formal verification process even more transparent to the user, the result of the model checking is back-annotated to the gamma language. This way, the result trace can be analyzed in a familiar domain which facilitates the easy correction of flaws.

Additionally, temporal logic expressions are automatically generated based on the composite system models, all of which describe a state reachability criterion. With the model checker these expressions can be used to test whether each specified state in the components of the composite system model is reachable, i.e., a test suite with full state coverage is generated.

Chapter 4

Language Extensions for Composition

The gamma constraint, statechart and interface languages presented in Section 3.2 were mainly designed by fellow students and researchers of the Fault Tolerant Systems Research Group and we defined only minor additions to it. This chapter introduces an extension to the gamma statechart language, which supports the definition of *hierarchical composite components*. The extension, referred to as composition language, is now integrated into the gamma language.

The goal of the composition language is to support the hierarchical composition of elementary statechart components introduced in Section 3.2. The language supports three composition types: *1) asynchronous-reactive* composition which is suitable for the modeling of distributed execution, *2) synchronous-reactive* for components of a single program or for highly synchronous communication, and *3) cascade* supporting the logical decomposition of a single operation. The following sections introduce the elements of the composition language as well as their precise semantics. Furthermore, an example is shown to summarize the elements of the gamma language. The section ends with a summary on how the functionalities of the framework have been and will be extended in accordance with the extensions of the gamma language.

4.1 The Composition Language

The gamma composition language supports the definition of communicating composite systems built from individual components. This is achieved by defining ports and interfaces, which enable individual components to act as communicating endpoints (Section 4.1.1). The communication between endpoints is supported by channels, responsible for connecting port instances (Section 4.1.2). Finally, the hierarchical composition of components is supported by various component types (Section 4.1.3). The following sections introduce the role of the elements of the gamma composition language in addition to their textual syntax and validation rules.

4.1.1 Endpoint Elements

This section introduces the *port* and *interface realization* elements of the gamma composition language. This part of the metamodel is depicted in Figure 4.1.

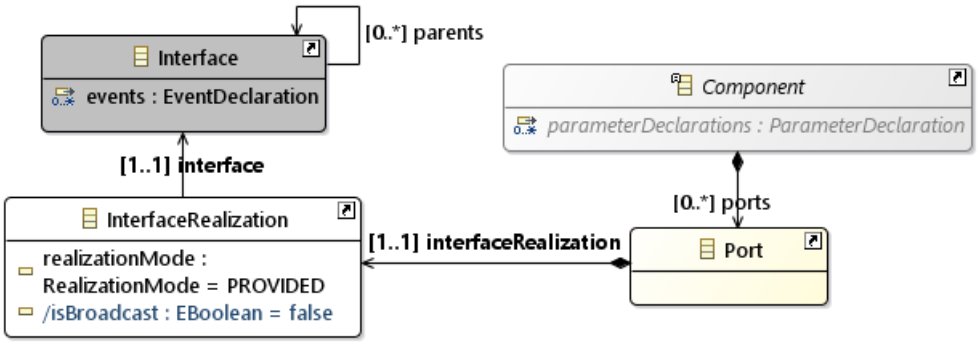


Figure 4.1: Fragment of the metamodel supporting the communication of components.

Port Ports serve as endpoints of component instances in a composite component model, through which events can be dispatched or received. Events are either called *signals* in case of synchronous components or *messages* in case of asynchronous components. Each port realizes a single interface by defining an interface realization. Communication between component instances, and between the composite component and its environment happens through ports.

Interface realization Interfaces can be realized in either *provided* or *required* mode. The difference is explained using the following interface definition:

```
interface InterfaceExample {
  in event a
  out event b
  inout event c
}
```

Note that *inout event c* is just a syntactic sugar for *in event c* and *out event c*.

- In provided mode: Ports dispatch and receive events according to the direction specified in the event declarations. In the current example, the component owning the realizing port will be able to dispatch events b and c (*out* events), and receive events a and c (*in* events) through this port. Such a port would be defined as:

  ```
  port providePort : provides InterfaceExample
  ```

- Interfaces are "turned inside out", that is, events declared with the direction *in* will be dispatched, and events declared with the direction *out* will be received through such ports. A component owning a port realizing *InterfaceExample* in required mode will be able to dispatch events a and c (declared *in* events in the interface), and receive events b and c (declared *out* events) through this port. Such a port would be defined as:

  ```
  port requirePort : requires InterfaceExample
  ```

Note that if two ports realize the same interface, one of them in provided mode, the other one in required mode, they can be connected since the direction of the events would match. Therefore, after connection they can exchange events with each other. As the example demonstrates, the realization mode does not specify a single direction in which events are transmitted through the particular port – dispatch and reception can be mixed in both cases.

We say that a port is a *broadcast* port if the interface realization mode is provided and the realized interface contains only *out* events. Unlike other ports, a broadcast port can be connected to multiple ports realizing the same interface in required mode with signal-based synchronous communication, since required ports are not able to dispatch events to the broadcast port, so no congestion will occur.

The concept of ports realizing interfaces in providing or requiring modes may be unusual to some designers, since ports usually support one-way event transmission in modern modeling languages. Our goal with this solution is to investigate the possibilities residing in interface-based communication in the domain of reactive systems. On the other hand, it is possible to use only out events on every interface – then provided mode is "output" mode and required mode is "input" mode.

4.1.2 Communication Elements

This section introduces the *instance port reference*, *port binding* and *channel* elements of the gamma composition language. This part of the metamodel is depicted by Figure 4.2.

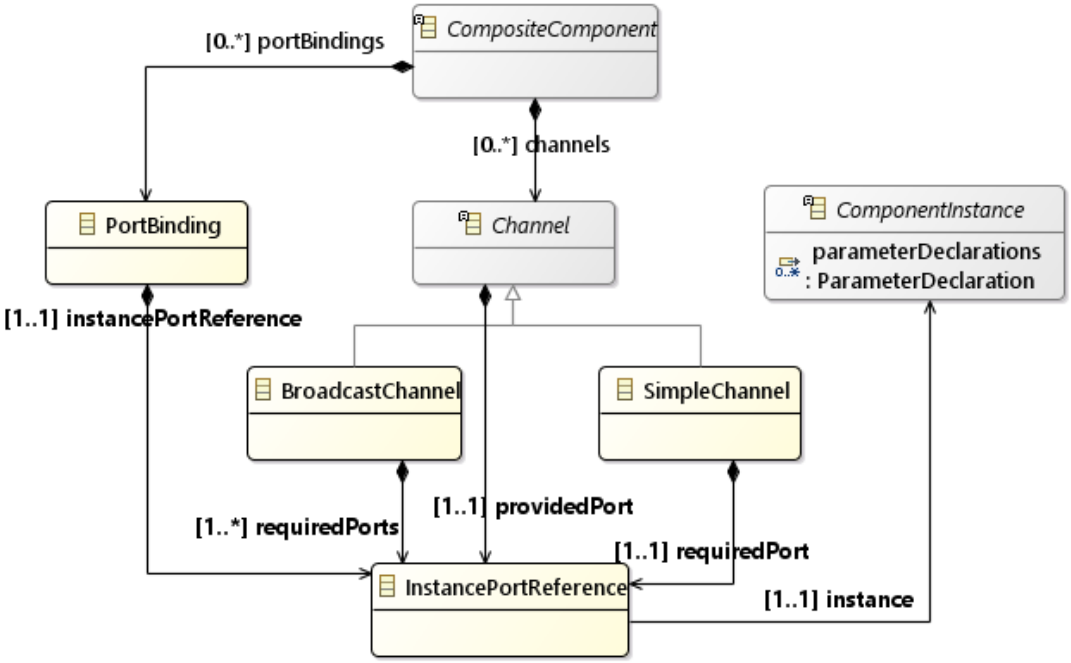


Figure 4.2: Fragment of the metamodel supporting the communication of components instances in composite components.

Instance port reference The instance port reference element is responsible for the identification of a port of a component instance, thus it refers to a single port and a single component instance.

Well-formedness constraint: the referred port must be contained by the type of the referred component instance.

Port binding The port binding element is responsible for the connection of the exterior ports of a composite component and the ports of constituent component instances. Therefore, it refers to a single port and a single instance port reference. Owing to this design, all events

received on the particular system port will be transmitted to the port of the associated component instance, and all the events dispatched through the particular instance port will be transmitted to the system port.

Well-formedness constraints: *1)* a port of a composite component must be connected to a port of a constituent component instance; *2)* the system level port and the port of the component instance must realize the same interface in the same realization mode.

A port binding that binds the port *systemPort* of a composite component to port *simplePort* of component instance *processor* can be defined as follows.

```
bind systemProvidedPort -> processor.simpleProvided
```

Channels are responsible for the connection of component instance ports. Technically, they use instance port references to refer to the endpoints. There are two types of channels: *broadcast* channel and *simple* channel.

- Simple channels support the connection of a single port providing and a single port requiring the same interface. As explained in Section 4.1.1, this design is valid and safe since they handle the same events with appropriate directions. A channel that connects port *simpleProvided* of *processor1* and *simpleRequired* of *processor2* can be defined as follows.

  ```
  channel [processor1.simpleProvided] -o)- [processor2.simpleRequired]
  ```

  Well-formedness constraint: in case of synchronous and cascade composite components a port must not be referred to in more than one channel or port binding (to avoid congestion, as synchronous components does not have message queues).

- Broadcast channels support sending events to multiple target ports. Such channels refer to *1)* a single broadcast port and *2)* multiple ports requiring the same interface as the one the broadcast port provides. In this case the direction of event transmission is determined: the broadcast port dispatches events and all the other ports connected to it receive them. A broadcast channel that connects broadcast port *broadcastProvided* of *processor1* to *simpleRequired* of *processor2* and *processor3* can be defined as follows.

  ```
  channel [processor1.broadcastProvided] -o)- [processor2.simpleRequired,
      processor3.simpleRequired]
  ```

  Well-formedness constraint: similarly to simple channels, required ports must not be referred to in more than one channel or port binding in case of synchronous and cascade composite components.

4.1.3 Composition Elements

This section introduces the *package* element as well as all the elements that are descendants of *component*.

Package Package is the root element of a composite component model. It supports the declaration of functions and constants in addition to the definition of one or more components. Furthermore, packages can depend on other packages by importing them. When a particular package is imported, its components, e.g., statechart definitions and composite components ,become visible from the importer package. Consequently, such imported components can be used as the type of component instances.

Well-formedness constraint: there must not be any circular dependencies between the packages.

Package *asyncCrossroad* with an imported package *crossroad* and a constant declaration *QUEUE_CAPACITY* can be defined as follows. Additionally, a synchronous component wrapper *AsyncCrossroadComponent* is defined.

```
import crossroad

package asyncCrossroad

    const QUEUE_CAPACITY : integer := 8
```

QUEUE_CAPACITY could be reused in any of the component definitions. If another package imported *main*, then the importer could refer to and reuse all the elements of the package.

Component Components serve as *types* of component instances in a composite component. They can be parameterized, that is, they can have parameters that can be referred to in their bodies, e.g., when specifying the attributes of a message queue. A single component can have any number of ports facilitating communication between components. Component is an abstract element; its descendants can be classified into *synchronous* and *asynchronous* components (see later in this section).

Composite component Composite components represent components that comprise of multiple *component instances* (synchronous or asynchronous). The integration of such component instances constitute the behavior of a particular composite component. The integration of components is supported by the elements introduced in Section 4.1.2.

Component instances are individual reactive elements with internal state, capable of receiving and dispatching events through ports. Each component instance refers to a single component, which serves as its type, that is, the component determines the ports on which the component instance shall be able to communicate as well as the internal states it shall be able to assume and the transitions it should be able to fire. Component instances can be either *atomic* or *composite*. The instance is atomic if the corresponding component is a statechart definition and composite if it refers to a composite component. This enables the hierarchical composition of systems.

Component instances can be either synchronous or asynchronous. Synchronous instances can refer to only synchronous components as their types, whereas asynchronous instances must refer to asynchronous components. A component instance *processor* with a type *Processor* can be defined as follows.

```
component processor : Processor
```

Synchronous components

This section introduces the *synchronous components* (*synchronous composite component* and *cascade composite component*). This part of the metamodel is depicted by Figure 4.3.

Synchronous component Synchronous components represent systems that communicate in a synchronous manner. There are three elements in the composition language that are synchronous components: statechart definitions (introduced in Section 3.2), synchronous composite components and cascade composite components. Synchronous components communicate

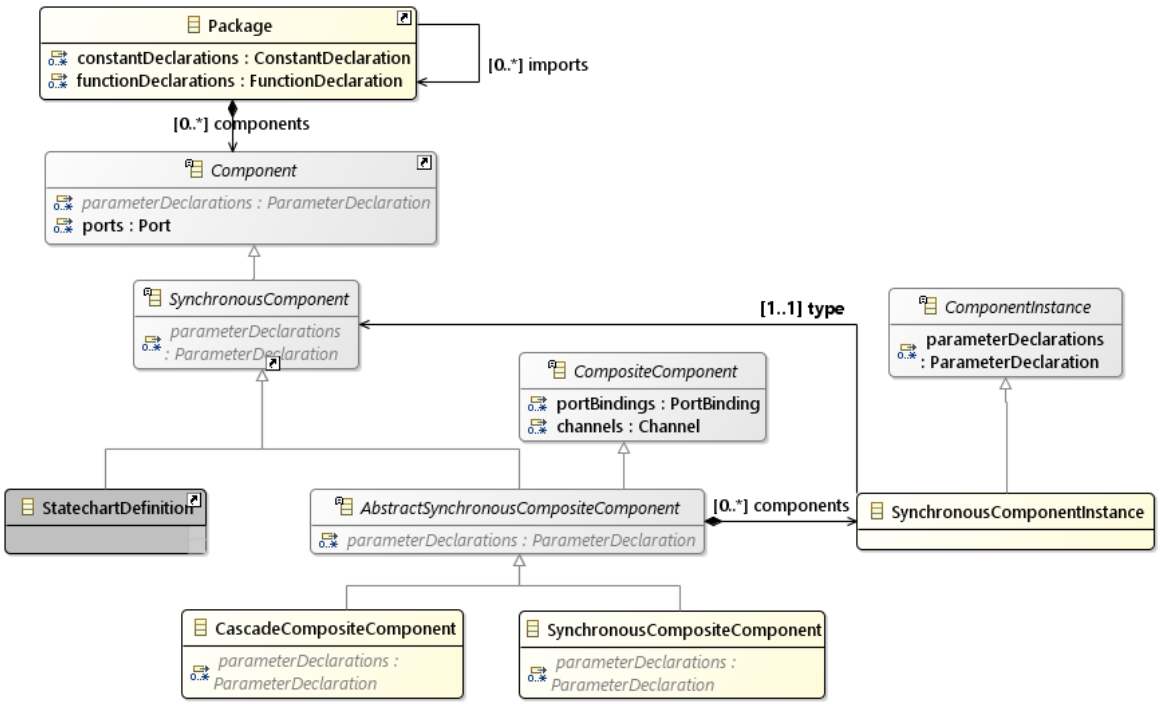


Figure 4.3: Fragment of the metamodel describing the synchronous component types.

with each other using *signals*. Synchronous components do not run independently, but their execution is scheduled by a scheduler: a wrapping asynchronous component or even a custom scheduler implementation. When executed, synchronous components process incoming signals and produce output signals in accordance with their internal states. Output signals are produced for a single execution period only, i.e., another execution might produce different output signals overwriting the output signals of the previous one.

Synchronous composite component The execution of a synchronous composite component conforms to a turn-based semantic. A turn is called a *cycle*. In each cycle all component instances of the particular composite component are executed. Although the order of the execution of the component instances is not defined, it is fixed, therefore they are executed in the same order in each cycle. This does not cause a loss of generality, as components *cannot affect* each other in a single execution cycle: if a component instance produces a signal that another component instance receives, the receiver will get it in the *next* cycle.

If a single component instance is executed it may *1)* process all signals received in the last execution turn, *2)* assume a new state according to the processed signals (new state configuration, new variable values) and *3)* produce signals that can be received by components (others or itself).

A synchronous composite component can be defined as follows. *TwoProcessors* has a single port *systemControlPort*. Additionally, it contains two component instances *processor1* and *processor2*, both referring to synchronous component *Processor* as type. The system level port *systemControlPort* is bound to *controlPort* of *processor1*. Also, *TwoProcessors* contains a channel that facilitates the communication between the component instances: the signals *processor1* transmits through port *protocol* will be received by *processor2* through *controlPort* and vice versa.

```
sync TwoProcessors [
    port systemControlPort : provides Control
] {
    component processor1 : Processor
    component processor2 : Processor

    bind systemControlPort -> processor1.controlPort

    channel [processor1.protocol] -o)- [processor2.controlPort]
}
```

Cascade composite component Cascade composite components contain the same elements as synchronous composite components, but their execution semantics is different. The execution of a cascade composite component also consist of cycles. In a single cycle all components of the particular composite component are executed according to a topological ordering based on channels. If a component is executed, it processes all incoming signals and produces signals in accordance with its internal state. However, the signals are processed in the *same execution cycle* by receivers, and not in the next one as it is specified in synchronous composite components. Therefore, in cascade composite components the flow of signals is one-way: component instances earlier in the ordering transmit signals to the ones that are later, in accordance with channel definitions. This structure ensures that a component can process all signals it receives in a single cycle.

Well-formedness constraints: *1)* every channel must be a broadcast channel (i.e., transmit events only one way), *2)* the contained component instances and their channel connections must form a *directed acyclic graph* (DAG), where the component instances are considered as nodes, and the direction of signal flow from one component to another is marked as a directed edge.

A cascade composite component can be defined as follows. *FourProcessors* is very similar to the synchronous composite component defined earlier. Additionally, it contains additional component instances *processor3* and *processor4* with type *Processor*. The execution of *FourProcessors* starts with *processor1*: it processes the incoming signals received through *systemControlPort* and transmits signals through port *protocol* to *processor2* and *processor4*. Next, *processor2* is executed, processing the incoming signals received through port *controlPort*, and transmits signals to *processor3* thorugh port *protocol*. Next, *processor3* is executed: it processes the signals it received through port *controlPort*. Finally, *processor4* is executed, processing the signals received through port *controlPort*. Note that a component instance never transmits signals to other instances defined earlier, ensuring the DAG configuration.

```
cascade FourProcessors [
    port SystemControlPort : provides Control
] {
    component processor1 : Processor
    component processor2 : Processor
    component processor3 : Processor
    component processor4 : Processor

    bind systemControlPort -> processor1.controlPort

    channel [processor1.protocol] -o)- [processor2.controlPort,
      processor4.controlPort]
    channel [processor2.protocol] -o)- [processor3.controlPort]
}
```

Asynchronous components

This section introduces *asynchronous component* elements *synchronous component wrapper* and *asynchronous composite component*. This part of the metamodel is shown in Figure 4.4.

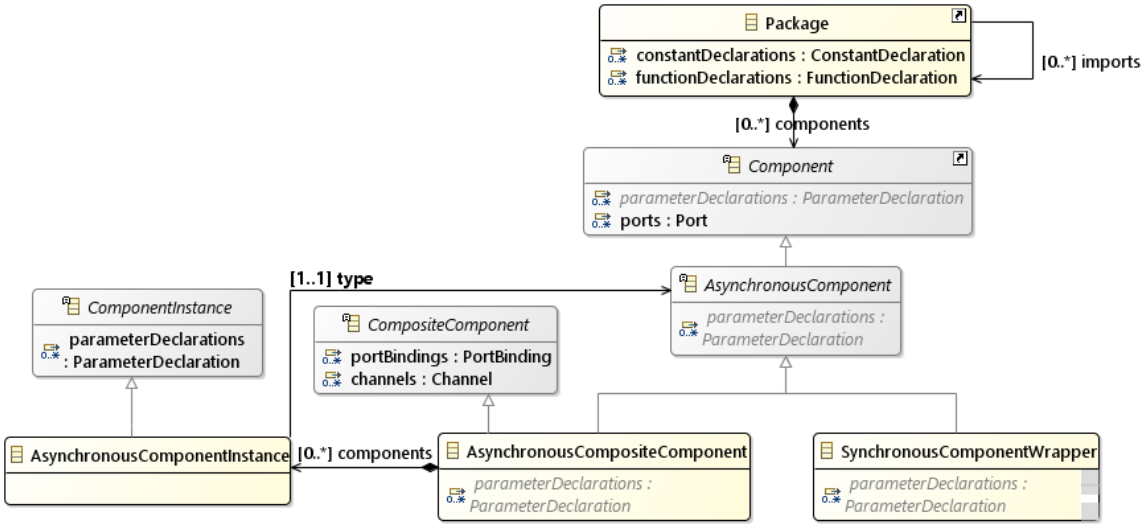


Figure 4.4: Fragment of the metamodel describing the asynchronous component types.

Asynchronous component Asynchronous components represent independently running component instances. There is no guarantee on the running time or the running frequency of such components. Asynchronous components communicate with each other via ports using buffered *messages*. There are two types of asynchronous components: asynchronous composite component and synchronous component wrapper.

Asynchronous composite components support the hierarchical definition of asynchronous components. Similarly to synchronous composite components, an asynchronous composite component consists of port bindings and channels in addition asynchronous component instances, which must refer to an asynchronous component as type. It is important to note that contained composite instances cannot have synchronous components as types in asynchronous composite components. In such cases, synchronous component wrappers can be used, which assign asynchronous behavior to synchronous components.

An asynchronous composite component can be defined as follows. *AsyncProcessor* contains a single port *systemControlPort*, two component instances *asyncProcessor1*, *asyncProcessor2* with type *ProcessorWrapper* and a port binding, which binds the system level *systemControlPort* to the *controlPort* of component instance *asyncProcessor1*.

```
async AsyncProcessor [
  port systemControlPort : provides Control
] {
  component asyncProcessor1 : ProcessorWrapper
  component asyncProcessor2 : ProcessorWrapper

  bind systemControlPort -> asyncProcessor1.controlPort
}
```

Synchronous component wrapper A synchronous component wrapper, depicted by Figure 4.5, wraps a single synchronous component, turning it into an asynchronous component. It is important to note that synchronous component wrappers implicitly have all ports of the wrapped synchronous component – additional defined ports are generally used for the reception of control messages. Furthermore, a synchronous component wrapper has one or more **message queues**, which store the incoming messages of the component. A message queue has multiple attributes:

- Capacity specifies the maximum number of messages that can be stored in the particular queue. If a queue is full and an additional message is received, the message is discarded. This attribute is essential in verification processes of composite components.
- Priority specifies the order in which the contents of message queues are retrieved during the execution of the asynchronous component. A message is always retrieved from a non-empty queue with the highest priority. Priority values can be any integers.
- Event references specify the types of messages that can be stored in the particular message queue. If a message arrives to an asynchronous component whose type is not associated to any of the contained message queues, then the event gets discarded. Therefore, it is always important to ensure that each incoming message type can be stored in a message queue. If a particular message could be stored in multiple message queues, the one declared first will be used (to enable hierarchical filters).

During execution, messages are retrieved *individually* from messages queues. A message is always taken from the highest priority non-empty queue. If the particular message was received on a port that is implicitly derived from the wrapped component, the message is converted to a signal (as synchronous components communicate with signals) and transmitted to the wrapped synchronous component (potentially overwriting previously sent signals). If it was received on a port explicitly defined on the wrapper component, the message does not get transmitted.

A synchronous component wrapper also has one or more **control specifications**, which specify the message types that are able to trigger the execution of the particular component. If a message with a specified type arrives to the wrapper component, the wrapped synchronous component may be executed in one of the following ways:

- Run once: the synchronous component executes a single cycle.
- Run to completion: the synchronous component executes as many cycles as needed to ensure all inner signals are processed and no additional steps could be taken.
- Reset: the synchronous component gets into its initial state.

Note that messages are transmitted to the wrapped component before execution, so control specifications can trigger on any event, including the ones defined by the wrapped component. Also note that the trigger must not be a complex trigger (see Section 3.2.1).

Finally, synchronous component wrappers can contain zero or more **clocks**, which emit tick events at defined timed intervals. Such time intervals can be defined with attribute *rate*. Currently, seconds and milliseconds are supported as unit of measurements. Tick events can be handled in control specifications similarly to regular events received from ports.

Example 1 (Control specifications). To demonstrate the flexibility of this control specification-based approach, we present two different execution semantics.

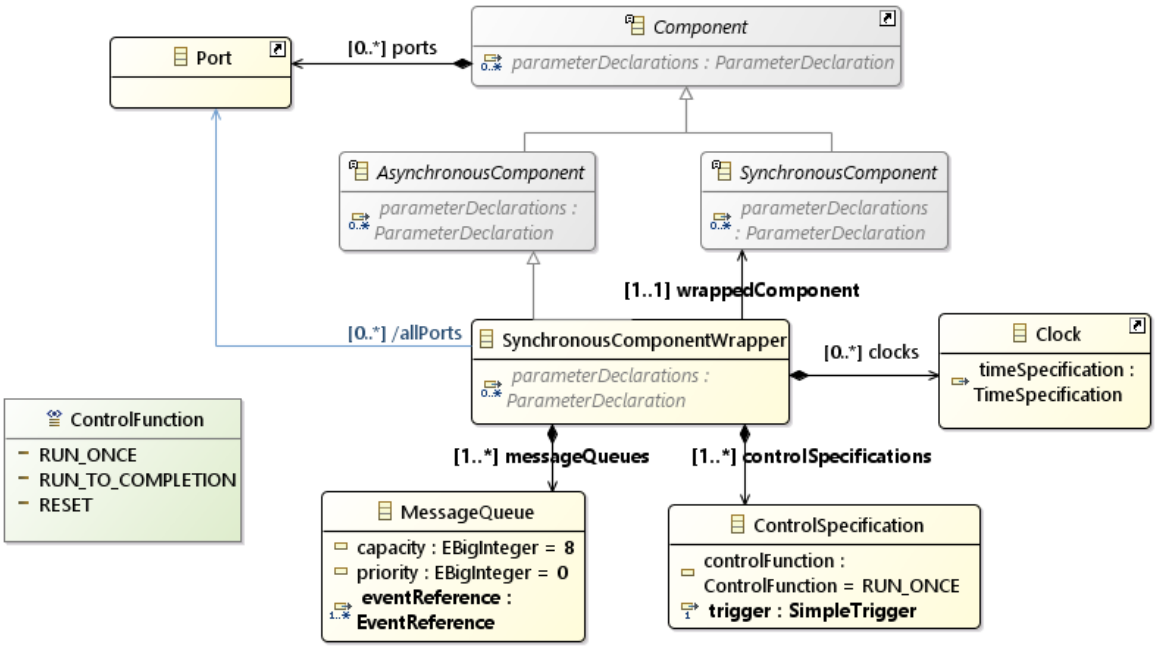


Figure 4.5: Fragment of the metamodel describing the synchronous component wrapper element.

- A single control specification is defined that triggers on the "*any event*". In this case, every time an event is retrieved from a message queue, the wrapped component gets executed. This behavior is similar to the semantics of UML statecharts.
- A single control specification is defined that triggers on the *ticks of a clock*. In this case, the wrapped component gets executed in defined periods of time and processes the events that arrive in between the actual and the previous clock tick.

Syntactically, a synchronous composite wrapper can be defined as follows. *ProcessorWrapper* wraps synchronous component *Processor* and has a single port *controlPort*. Also, it contains a single clock *millisecondClock* with rate 10 ms; a control specification, defining that event *execute* from port *controlPort* shall trigger a *single cycle* execution; and a message queue *controlPortMessages* with priority 1 and capacity 8 that is able to store any kind of message received through port *controlPort*.

During execution, messages coming in to *controlPortMessages* are converted to signals and transmitted to the wrapped component *Processor*. When an *execute* message is observed, it is converted and transmitted to the wrapped component *Processor*, which is followed by the initiation of a single execution cycle on the wrapped component *Processor*.

```
async ProcessorWrapper of Processor [
    port controlPort : provides Control
] {
    clock millisecondClock(rate = 1 ms)

    when controlPort.execute / run

    queue controlPortMessages (priority = 1, capacity = 8) {
        controlPort.any
    }
}
```

4.1.4 Summary

As a summary, Table 4.1 describes the component types that are supported by the gamma composition language in terms of *synchronousness* and *compositeness*.

Table 4.1: Component types supported by the gamma composition language.

	Atomic	**Composite**
Synchronous	Statechart	Synchronous composite component Cascade composite component
Asynchronous	Synchronous component wrapper	Asynchronous composite component

Figure 4.6 presents the containment hierarchy of an example composite model. Note that the synchronous and asynchronous domain can be bridged only by synchronous composite wrappers. Furthermore, the leaves, which generally define the behavior of composite components, are always statechart definitions.

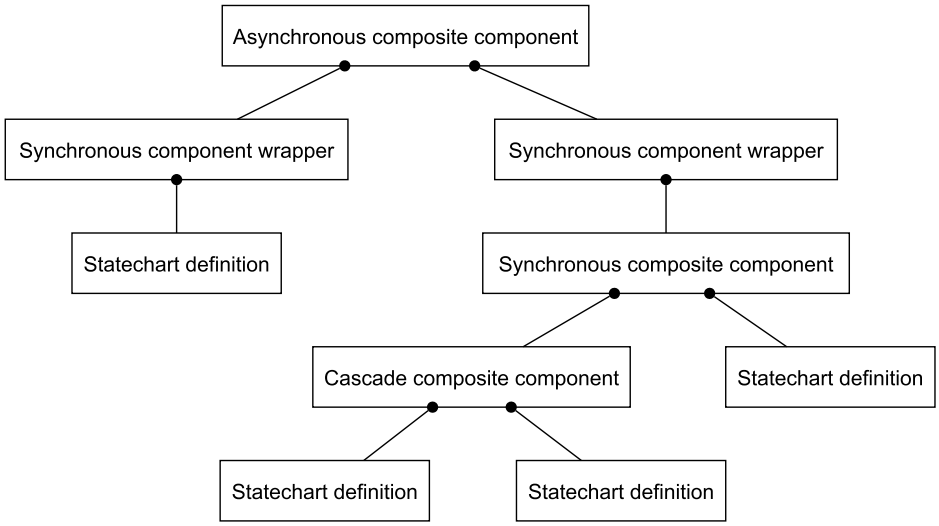


Figure 4.6: A composite model hierarchy in gamma.

4.2 Composition Semantics

As presented in Section 4.1, the gamma framework supports several composition semantics. To ensure unambiguity, this section formally introduces the semantics of each composition mode, on which both source code generation and model transformations of the framework heavily rely. The section starts with the introduction of synchronous component and synchronous composite component semantics, which are followed by the semantics of the cascade composite component (Section 4.2.1). Finally, asynchronous component and synchronous component wrapper semantics are presented (Section 4.2.2).

4.2.1 Synchronous Components

Synchronous components represent systems communicating in a synchronous manner. Their behavior is close to that of deterministic and fully specified finite state machines, introduced in Section 2.2. The main difference is that in case of synchronous components events

can be *parameterized*; an *event instance* consists of an *event type* and a *parameter value* in accordance with the domain of the particular event. It is important to note that in case of synchronous components, events can be *absent*.

Definition 2. A synchronous component is a tuple $C_s = (S, s^0, I, O, \mathcal{D}, T)$, where:

- S is the set of potential states, with $s^0 \in S$ being the initial state.
- I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$.
- $\mathcal{D} : I \cup O \to \{d_1, \ldots, d_n\}$ is a function that assigns a finite domain to every input and output event. We say that an event $e \in I \cup O$ is *parameterized* if $|\mathcal{D}(e)| > 1$.
- An *event instance* is a pair of an event and a parameter (e, p) where $e \in I \cup O$ and $p \in \mathcal{D}(e)$. The set of all event instances for a given event e is denoted by $inst(e) = \{(e, p) \mid p \in \mathcal{D}(e)\} \cup \{(e, \perp)\}$, where $(e, \perp)$ denotes the absence of the event.
- An *input vector* v_I is a function that assigns an event instance to every input event $e \in I$: $v_I(e) \in inst(e)$. Similarly, an *output vector* v_O is a function that assigns an event instance to every output event $e \in O$: $v_O(e) \in inst(e)$. The sets of all possible input and output vectors are denoted by V_I and V_O, respectively.
- T: $S \times V_I \to S \times V_O$ is the transition function, which determines the next state and the output vector of the component when executing it in a given state with a given input vector. Note that this definition requires the component to have a deterministic behavior. We also require T to be fully specified.

A **synchronous composite component** consists of several synchronous components. It can receive input events from its environment (visible input events), all of which are transmitted to its constituents for processing. Visible output events of the synchronous composite component are also produced by contained components. Furthermore, the communication of contained components is supported by *channels*, which transmit output events of particular components to other ones as input events.

Definition 3. A synchronous composite component is a tuple $\mathcal{C}_s = (\mathbf{C}, I, O, \mathbb{C})$, where:

- $\mathbf{C} = \{(S_1, s_1^0, I_1, O_1, \mathcal{D}_1, T_1), \ldots, (S_K, s_K^0, I_K, O_K, \mathcal{D}_K, T_K)\}$ is the set of synchronous components constituting the composite component.
- $I \subseteq \hat{I}$ is the set of visible input events, where $\hat{I} = \bigsqcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of visible output events, where $\hat{O} = \bigsqcup_{k=1}^{K} O_k$.
- $\mathbb{C} : \hat{I} \setminus I \to \hat{O}$ is the *channel* function that assigns an output as the source of events to every input, with the restriction that it must not be defined for elements of I, that is, an input is either linked to an output or is a visible input.

As the metamodel fragment in Figure 4.3 presents, synchronous composite components are synchronous components. The possible states of a synchronous composite component are implicitly determined by its constituents. The execution of a synchronous composite component is turn-based, a turn is called a *cycle*. In each cycle all component instances are executed: they process incoming signals and produce outgoing ones. It is import to note that components *do not affect* each other in a single execution cycle: if a component instance transmits a signal to another component via a channel, the receiver will process it in the *next* cycle.

Definition 4. A synchronous composite component $\mathcal{C}_s$ is itself a synchronous component $C_s(\mathcal{C}_s) = (S, s^0, I, O, \mathcal{D}, T)$, where:

- $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ is the set of potential states, derived as all possible combinations of the potential states of the constituent synchronous components and the last output events of every component.
- $s^0 = (s_1^0, \ldots, s_K^0, \perp_{\hat{O}})$ is the initial state, where every constituent synchronous component is in its initial state and the last output vector $\perp_{\hat{O}} \in V_{\hat{O}}$ assigns $\perp$ to every output event ($\forall e \in \hat{O} \ : \ \perp_{\hat{O}}(e) = \perp$).
- I is the set of input events as defined in Definition 3.
- O is the set of output events as defined in Definition 3.
- $\mathcal{D}$ is implicitly defined by $\mathcal{D}_k$, as $\hat{\mathcal{D}} = \bigsqcup_{k=1}^{K} \mathcal{D}_k$ and $\mathcal{D}(e) = \hat{\mathcal{D}}(e)$ for all $e \in I \cup O$.
- The transition function is defined as $T((s_1, \ldots, s_K, v_{\hat{O}}), v_I) = ((s'_1, \ldots, s'_K, v'_{\hat{O}}), v_O)$, where:
  - For each input event $e \in \hat{I}$ of any constituent component let $v_{\hat{I}}(e) = v_I(e)$ if $e \in I$ or $v_{\hat{I}}(e) = v_{\hat{O}}(\mathbb{C}(e))$ otherwise. Note that $v_{\hat{I}}$ implicitly defines every v_{I_k} as well, because $v_{\hat{I}} = \bigsqcup_{k=1}^{K} v_{I_k}$.
  - The next state s'_k of every component corresponds to the transition function T_k such that $T_k(s_k, v_{I_k}) = (s'_k, v'_{O_k})$.
  - $v'_{\hat{O}} = \bigsqcup_{k=1}^{K} v'_{O_k}$ is the new set of "previous outputs".
  - The output of the synchronous composite component for each visible output event $e \in O$ is defined by the output of the constituent components: $v_O(e) = v'_{\hat{O}}(e)$.

Cascade composite components are very similar to synchronous composite components. The difference is that in case of cascade components, the contained components are *ordered*. A component can transmit signal to another one via a channel, only if it is defined earlier in the ordering than the other one.

Definition 5. A cascade composite component is a tuple $\mathcal{C}_c = (\mathbf{C}, I, O, \mathbb{C})$, where:

- $\mathbf{C} = ((S_1, s_1^0, I_1, O_1, \mathcal{D}_1, T_1), \ldots, (S_K, s_K^0, I_K, O_K, \mathcal{D}_K, T_K))$ is an *ordered sequence* of synchronous components constituting the cascade composite component.
- $I \subseteq \hat{I}$ is the set of visible input events, where $\hat{I} = \bigsqcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of visible output events, where $\hat{O} = \bigsqcup_{k=1}^{K} O_k$.
- $\mathbb{C} : \hat{I} \setminus I \to \hat{O}$ is the *channel* function that assigns an output as the source of events to every input except the visible ones, that is, an input is either linked to an output or is a visible input. Furthermore, if $\mathbb{C}(e_1) = e_2$ such that $e_1 \in I_k$ and $e_2 \in O_l$ then we require $l < k$.

Similarly to synchronous composite components, a cascade composite component is also a synchronous component. The possible states of a cascade component are implicitly determined by the contained components. The execution of a cascade composite component also consists of cycles. In a single cycle all contained components of the particular cascade component are executed in the *order of their definition*. If a component is executed, it processes all incoming signals and produces outgoing signals. However, the produced signals are processed in the *same execution cycle* by receivers, and *not in the next one* as it is specified in synchronous composite components.

Definition 6. A cascade composite component $\mathcal{C}_c$ is itself a synchronous component $C_s(\mathcal{C}_c) = (S, s^0, I, O, \mathcal{D}, T)$, where:

- $S = S_1 \times \ldots \times S_K$ is the set of potential states, derived as all possible combinations of the potential states of the constituent synchronous components.
- $s^0 = (s_1^0, \ldots, s_K^0)$ is the initial state, where every constituent synchronous component is in its initial state.
- I is the set of input events as defined in Definition 5.
- O is the set of output events as defined in Definition 5.
- $\mathcal{D}$ is implicitly defined by $\mathcal{D}_k$, as $\hat{\mathcal{D}} = \bigsqcup_{k=1}^{K} \mathcal{D}_k$ and $\mathcal{D}(e) = \hat{\mathcal{D}}(e)$ for all $e \in I \cup O$.
- The transition function is defined as $T\big((s_1, \ldots, s_K), v_I\big) = \big((s'_1, \ldots, s'_K), v_O\big)$, computed iteratively:
  - For the first component $(S_1, s_1^0, I_1, O_1, \mathcal{D}_1, T_1)$, Definition 5 implies that $I_1 \subseteq I$. Therefore, we can apply $T_1(s_1, v_{I_1}) = (s'_1, v_{O_1})$ to compute s'_1 and v_{O_1}, where $v_{I_1}(e) = v_I(e)$ for every $e \in I_1$.
  - For subsequent components $(S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$, the input vector v_{I_k} will be defined by the visible input events and the output vectors of previously executed components: $v_{I_k}(e) = (v_I \cup \bigsqcup_{l=1}^{k} v_{O_l})(e)$ due to Definition 5. Therefore, we can apply $T_k(s_k, v_{I_k}) = (s'_k, v_{O_k})$ to compute s'_k and v_{O_k}.
  - After the iteration, the result of $T\big((s_1, \ldots, s_K), v_I\big)$ is $\big((s'_1, \ldots, s'_K), v_O\big)$, where $v_O(e) = \big(\bigsqcup_{k=1}^{K} v_{O_k}\big)$ for every $e \in O$.

4.2.2 Asynchronous Components

Asynchronous components represent component instances, which are able to run independently. As opposed to synchronous components, the absence of events *is not interpreted* in case of asynchronous components. Furthermore, an asynchronous component is always triggered on a *single* input event, and, opposed to the state machine formalism in Section 2.2, capable of producing a *sequence* of output events. An asynchronous component is deterministic in terms of state changes, but non-determinism is possible regarding the ordering of output events.

Definition 7. An asynchronous component is a tuple $C_a = (S, s^0, I, O, \mathcal{D}, T)$, such that:

- S is the set of potential states, with $s^0 \in S$ being the initial state.
- I is the set of input events and O is the set of output events such that $I \cap O = \emptyset$.
- $\mathcal{D} : I \cup O \to \{d_1, \ldots, d_n\}$ is a function that assigns a finite domain to every input and output event. We say that an event $e \in I \cup O$ is *parameterized* if $\mathcal{D}(e) > 1$.
- An *event instance* is a pair of an event and a parameter (e, p) where $e \in I \cup O$ and $p \in \mathcal{D}(e)$. The set of all event instances for a given event e is denoted by $inst(e) = \{(e, p) \mid p \in \mathcal{D}(e)\}$ (notice that contrary to Definition 3, there is no symbol for the absence of events).
- The set of all possible event instances is denoted by $E = E_I \cup E_O$, where $E_I = \bigsqcup_{e \in I} inst(e)$ and $E_O = \bigsqcup_{e \in O} inst(e)$ are the sets of all possible input and output event instances, respectively.

- $T \subseteq S \times E_I \times S \times \{E_O^*\}$ is the transition function, which determines the next state and the sequence of output events (E_O^*) of the component when executing it in a given state with a given input event. Note that this definition *does not* require the component to have a deterministic behavior. We still require T to be fully specified.

A **synchronous component wrapper** wraps a single synchronous component. In addition to the events the wrapped synchronous component is able to receive, the synchronous wrapper is capable of receiving events for control; they are called *control events*. Additionally, the synchronous wrapper defines *trigger predicates* (referred to as control specifications in Section 4.1), which specify the events that trigger the execution of the wrapped synchronous component.

Definition 8. An asynchronous wrapper for a synchronous component is defined as a tuple $A = (C_s, e_c, trig)$, where:

- $C_s = (S_s, s_s^0, I_s, O_s, \mathcal{D}_s, T_s)$ is the wrapped synchronous component.
- e_c is the *control event*.
- $trig : I_S \cup \{e_c\} \to \{\top, \bot\}$ is the *trigger predicate* that given an incoming event will return if the underlying synchronous component should be executed or not.

Synchronous component wrappers are asynchronous components. The possible states of a synchronous component wrapper are implicitly defined by the wrapped component and the message queues, capable of storing incoming events. During execution, the wrapper retrieves events individually from the message queues (always from the highest priority non-empty one) and transmits them to the wrapped component. The wrapped synchronous component stores these events until it gets executed. Also, if the last retrieved event from the message queue is considered as a trigger event by the trigger predicate, the synchronous component gets executed.

Definition 9. An asynchronous wrapper A for a synchronous component is itself an asynchronous component $C_a(A) = (S, s^0, I, O, \mathcal{D}, T)$, such that:

- $S = S_s \times \{E_I^*\}$ is the set of potential states, each state composed of a state of the wrapped synchronous component and a finite sequence of input event instances.
- $s^0 = (s_s^0, \varepsilon)$, where ε is the empty sequence.
- $I = I_s \cup \{e_c\}$ is the set of input events including the input events of the wrapped synchronous component and the control event.
- $O = O_s$ is the set of output events defined in the wrapped synchronous component.
- $\mathcal{D} = \mathcal{D}_s \cup (e_c \to \{d\})$ is the domain function of the wrapped synchronous component extended with a mapping that assigns a singleton set to the control event (indicating that it is not parameterized). The definition of *inst* should also be extended for e_c according to the definition of $\mathcal{D}$, as well as E_I as the set of potential event instances.
- The transition function is defined as $T\big((s_s, q), (e, v)\big) = \{(s_s', q')\} \times \Omega$, such that:
  - If $trig(e) = \bot$, then $s_s' = s_s$ and $q' = q \frown (e, v)$, where $\frown$ denotes concatenation. $\Omega = \{\varepsilon\}$ as the set of possible output sequences is only the empty sequence in this case.

– If $trig(e) = \top$, then s'_s should be such that $T_s(s_s, v_I(q)) = (s'_s, v_O)$, where

$$v_I\big((e_1, v_1), \ldots, (e_n, v_n)\big)(e) = \begin{cases} (e, \bot), & \text{if } \nexists i \ : \ e = e_i, \\ (e, v_i) & \text{otherwise, such that } \nexists j > i \ : \ e = e_j \\ & \text{(that is, the last event instance} \\ & \text{matching } e \text{ in the sequence)} \end{cases}$$

and $q' = \varepsilon$. $\Omega = \{\Sigma(v_O)\}$ as the set of possible output sequences is every possible permutation $\sigma(v_O)$ of the elements of the output vector.

Asynchronous composite components contain one or more asynchronous component instances of type asynchronous composite component or synchronous component wrapper. Dispatched messages are forwarded to the corresponding component instances with respect to causality rules, the parent asynchronous composite component does not store or delay them. Note that only synchronous wrappers are capable of storing messages for a longer period of time. Therefore, dispatched messages ultimately land in a message queue of a synchronous component wrapper.

4.3 Crossroads Example

This section introduces the usage of the gamma language through a simple example. This example defines a simple synchronous composite component model, called *Crossroads*, which can serve as a controller unit of traffic lights at a crossroad. The synchronous component model is going to contain two traffic light components as well as a controller component that controls the traffic light components. This synchronous component is going to be wrapped by a synchronous composite wrapper, enabling its independent execution.

Interfaces First of all, the interfaces have to be defined, which will be realized by the ports of the components in the composite component model.

```
interface LightCommands {
    out event displayRed
    out event displayGreen
    out event displayYellow
    out event displayNone
}
interface PoliceInterrupt {
  out event police
}
```

```
interface Control {
  out event toggle
}
interface Executable {
  out event execute
}
```

Statechart components After the interfaces have been defined, the statecharts interacting in the synchronous composite component model can be designed. In this example we have two statechart definitions: *TrafficLight* and *Controller*. *TrafficLight* models a standard 3-phase light looping through the red-green-yellow-red sequence. As an extra, *TrafficLight* supports an interrupted mode which may be triggered by the police – in this state, the traffic light is blinking in yellow. Statechart *Controller* models a unit, which is able to control two traffic lights, called *priority* and *secondary*, at a crossroad.

The composite component models that can be designed with the gamma framework heavily rely on the concept of *encapsulation*, that is the definition of components are not regarded

when component types (component declarations) are instantiated and connected, only ports of components are considered. Therefore, during the design of a composite component the users do not need to deal with the definition of components. The relevant part of statechart definition *TrafficLight* is as follows.

```
package trafficLight // Package of the statechart
    statechart TrafficLight [
        // External ports of component type TrafficLight,
        port lightCommands : provides LightCommands,
        port policeInterrupt : requires PoliceInterrupt,
        port control : requires Control
    ] {
        // Statechart definition
    }
```

The relevant part of statechart definition *Controller* is as follows.

```
package controller // Package of the statechart
    statechart Controller [
        // External ports of component type Controller
        port priorityControl : provides Control,
        port secondaryControl : provides  Control,
        port policeInterrupt : requires  PoliceInterrupt,
        port priorityPolice : provides  PoliceInterrupt,
        port secondaryPolice : provides  PoliceInterrupt
    ] {
        // Statechart definition
    }
```

Synchronous composite component The synchronous composite component definition *Crossroad*, composing two traffic lights and a controller, is as follows.

```
import trafficLight
import controller

// Package of the synchronous composite component model
package crossroad
  sync Crossroad [
    // External ports of the composite component model
    port police : requires PoliceInterrupt,
    port priorityLightOutput : provides LightCommands,
    port secondaryLightOutput : provides LightCommands
  ] {
    // Component instances
    component controller : Controller
    component priorityLight : TrafficLight
    component secondaryLight : TrafficLight
    // Bindings connecting system ports to internal ports
    bind police -> controller.policeInterrupt
    bind priorityLightOutput -> priorityLight.lightCommands
    bind secondaryLightOutput -> secondaryLight.lightCommands
    // Channel definitions connecting internal ports
    channel [controller.priorityControl] -o)- [priorityLight.control]
    channel [controller.secondaryControl] -o)- [secondaryLight.control]

    channel [controller.priorityPolice] -o)- [priorityLight.policeInterrupt]
    channel [controller.secondaryPolice] -o)- [secondaryLight.policeInterrupt]
  }
```

The synchronous composite model definition starts with the imports of packages *trafficLight* and *controller*. These packages contain the statechart definitions presented above, which are going to be used later in the component instance definition phase. The external ports of the composite component are *police*, realizing interface *PoliceInterrupt* in required mode, along with *priorityLightOutput* and *secondaryLightOutput*, both realizing interface *LightCommands* in provided mode. The composite definition, as introduced in the previous subsection, contains component instances, port bindings and channel definitions.

The defined component instances are *priorityLight* and *secondaryLight* both with type *TrafficLight*, modeling two traffic lights at a crossroad, as well as *controller* with type *Controller*.

External ports of the composite component models should be connected to ports of internal component instances, to enable interactions between the composite component and its environment. In this case, external port *police* is bound to port *PoliceInterrupt* of component *controller*. Additionally, external ports *priorityLightOuput* and *secondaryLightOutput* are bound to port *LightCommand* of component *priorityLight* and *secondaryLight*, respectively.

Finally, the channels are defined, which connect the ports of contained component instances, making their interactions possible. Port *priorityControl* and *secondaryControl* of component *controller* are connected to port *control* of *priorityLight* and *secondaryLight*, respectively. Similarly, port *priorityPolice* and *secondaryPolice* of component *controller* are connected to port *policeInterrupt* of *priorityLight* and *secondaryLight*, respectively. Note that this model does not contain broadcast channels.

Synchronous component wrapper Now a synchronous component wrapper can be used to wrap *CrossroadComponent* and enable its independent execution. The synchronous component wrapper is as follows.

```
import crossroad

package asyncCrossroad // Package of the wrapper
    const QUEUE_CAPACITY : integer := 8

    async AsyncCrossroadComponent of CrossroadComponent [
        // External port of the wrapper
        port execution : requires Executable
    ] {
        // Clock
        clock clockSignal(rate = 100 ms)
        // Control specifications
        when execution.execute / full step
        when clockSignal / run
        when police.interrupt / run
        // Message queues
        queue executionMessages (priority = 2, capacity = QUEUE_CAPACITY) {
            execution.execute, clockSignal
        }
        queue crossroadsMessages (priority = 1, capacity = QUEUE_CAPACITY) {
            police.any, priorityLightOutput.any, secondaryLightOutput.any
        }
    }
```

AsyncCrossroadComponent defines a single port *execution* on which the execution commands are expected. Messages received on port *execution* are stored in *executionMessages*, whereas the messages of additional (implicit) ports are stored in *crossroadsMessages*. The priority of the former is higher. Owing to the control specifications, when a message *execute* is retrieved, the wrapped component gets executed for a full step, that is, as many cycles as needed to

process all signals of each contained components of the wrapped component. When either a clock signal of *clockSignal* or an *interrupt* signal of port *police* is received, the wrapped component is executed for a single cycle.

The graphical representation of the designed composite model is depicted in Figure 4.7.

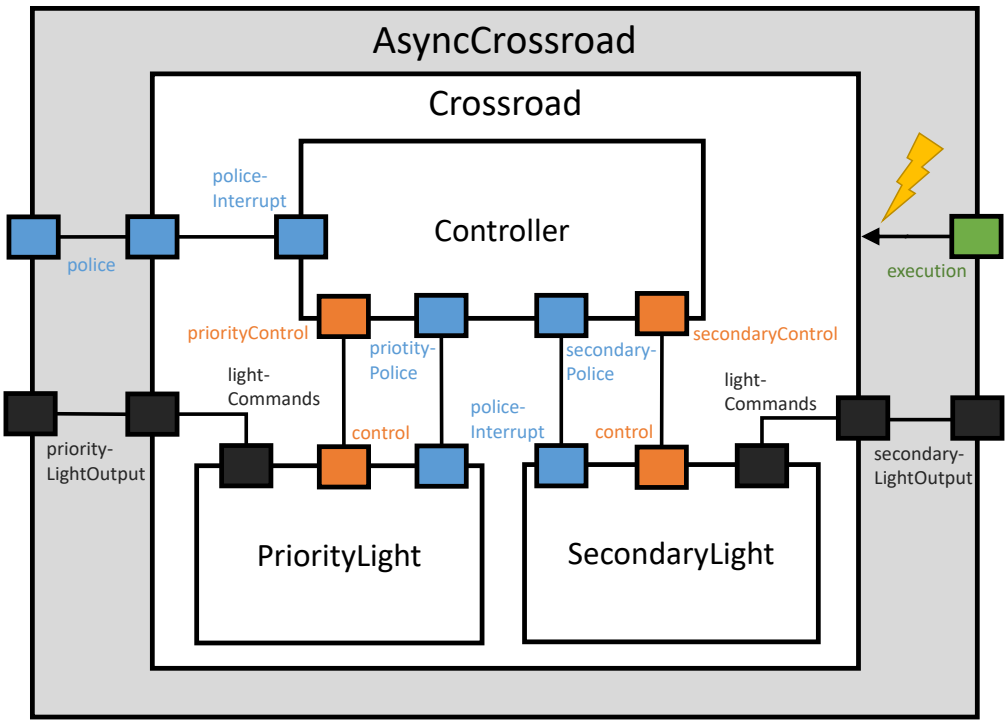


Figure 4.7: A graphical representation of the AsyncCrossroad gamma composite component model.

4.4 Extending the Code Generation Functionality

As it was presented in Section 4.1, the gamma language has been extended with several new elements, such as different kinds of composite components. To make them useful to the users, it is important to extend the functionalities of the framework as well, so they are capable of handling them. This section introduces the extensions and plans we have made to the Java source code generator of the gamma framework in order to support code generation for composite models. Currently, the Java code generator is capable of generating functionally working code from interfaces and synchronous components. However, the code snippets for the asynchronous components are not generated by the gamma framework. At this phase, we already have detailed plans about how the code generator (and the UPPAAL model transformation) should work with the asynchronous models, and this is an excerpt of the sample code that will guide the implementation.

4.4.1 Interfaces

Each gamma interface used in a composite component is transformed to a Java interface. Each generated Java interface contains three inner interfaces: *Listener*, *Provided* and *Required*. Inner interface *Provided* is realized by Java classes that represent ports realizing the particular gamma interface in provided mode. Similarly, interface *Required* is realized by classes that

represent ports realizing the particular gamma interface in required mode. Additionally, *Listener* contains two interfaces, also called as *Provided* and *Required*, which describe contracts for listeners of certain event dispatches. The following snippet describes the Java interface that is generated from the gamma interface *Executable*, presented in Section 4.3.

```
public interface ExecutableInterface {
  interface Listener {
    interface Provided {
      void raiseExecute();
    }
    interface Required {}
  }
  interface Provided extends Listener.Required {
    public boolean isRaisedExecute();
    void registerListener(Listener.Provided listener);
  }
  interface Required extends Listener.Provided {
    void registerListener(Listener.Required listener);
  }
}
```

Inner Java interface *Listener.Provided* contains a raising method for each event that can be sent by a port realizing the particular gamma interface in provided mode. For example, gamma interface *Executable* has a single out event *execute*, thus, a port realizing it in provided mode is able to dispatch *execute* events, which is indicated by raising method *raiseExecute* of the corresponding Java interface. If the gamma event had parameter declarations, the corresponding Java method would have the necessary parameters as well. Similarly, *Listener.Required* contains a raising method for each event that can be sent by a port that realized the particular gamma interface in required mode. In this example, as gamma interface *Executable* does not contain any in events, no method is contained by Java interface *Listener.Required*.

Java interface *Provided* extends *Listener.Required*. This ensures that each event that can be sent by a port realizing the particular interface in required mode (indicated by *Listener.Required*) can be *accepted* by any other port realizing it in provided mode (indicated by *Provided*). As *Listener.Required* is empty, *Provided* does not contain any method that would indicate event reception. Also, interface *Provided* contains "is raised" methods (*isRaisedExecute* in this example), which can be used to check whether a certain event has been dispatched recently: the last cycle in case of synchronous components and a defined period of time in case of asynchronous components. Interface *Provided* contains an additional method *registerListener*, which supports the registration of listener objects. When a port dispatches a particular event, the corresponding "raise" methods of the registered listener objects are called. Inner Java interface *Required* is very similar to *Provided*; the difference is that it turns interfaces the other way around as described in Section 4.1, otherwise all its functionalities are analogous to interface *Provided*.

Note that if a Java class represents a port realizing a particular interface in provided mode (that is it extends inner Java interface *Provided*), and another Java class represents a port of the same interface in required mode (extends Java interface *Required*), their instances can be connected very easily. Both has to be registered to the other one using the *registerListener* method. After that, they can automatically dispatch events to each other using the methods described by *Listener.Required* and *Listener.Provided*.

4.4.2 Components

Each component that has been used in a composite component is transformed to a Java class. The generated Java classes are slightly different when the corresponding component is synchronous or an synchronous wrapper or an asynchronous component. Moreover, in case of synchronous components, the implementation of composite and atomic components is also slightly different.

Synchronous components

A class representing an atomic component has the following fields: a statechart object implementing the behavior of the component and port objects representing gamma ports. Classes generated from composite components also contain port objects in addition to component objects representing the contained gamma components.

Composite components The following snippet describes some elements of the Java class generated from the gamma synchronous composite component *Crossroads*, presented in Section 4.3.

```
public class Crossroad {
  // Component instances
  private TrafficLight secondary = new TrafficLight();
  private TrafficLight priority = new TrafficLight();
  private Controller controller = new Controller();
  // Port instances
  private Police police = new Police();
  private PriorityLightOutput priorityLightOutput =
      new PriorityLightOutput();
  private SecondaryLightOutput secondaryLightOutput =
      new SecondaryLightOutput();
  //...
  /* Creates the channels between component instances */
  private void init() {
    controller.getSecondaryControl()
      .registerListener(secondary.getControl());
    secondary.getControl()
      .registerListener(controller.getSecondaryControl());
    controller.getSecondaryPolice()
      .registerListener(secondary.getPoliceInterrupt());
    secondary.getPoliceInterrupt()
      .registerListener(controller.getSecondaryPolice());
    //...
  }
  /* Executes a single cycle */
  public void runCycle() {
    changeComponentEventQueues();
    secondary.runCycle();
    priority.runCycle();
    controller.runCycle();
  }
  /* Executes as many cycles as needed to ensure
   that all generated events get processed */
  public void runFullCycle() {
    do {
      runCycle();
    }
    while (!hasInnerEvent());
  }
```

The fields represent contained component instances as well as ports used for communication with the environment as described in Section 4.4.1. The classes of the ports are defined as inner classes.

Method *init* is responsible for creating the channels, that is, it registers the corresponding ports of components as *listeners* using the *registerListener* method of ports. Furthermore, getter methods are generated that return the contained port instances, supporting event dispatch from the environment. This way the registration of unique listeners is also supported, thus, users can be notified about the occurrence of certain events.

The component can be executed by calling either its *runCycle* or *runFullCycle* method. Method *runCycle*, implementing the *run once* action. consists of the following steps.

1. It changes the event queues of the contained components.
2. It initiates a single cycle on each contained component.

As mentioned in Section 4.1, in case of synchronous composite components, the contained components cannot affect each other in a single cycle, but they process the received events at the beginning of the next cycle. This is implemented with a pair of queues: one storing the events of the actual cycle (events under process), and another one, storing the incoming events for the next cycle. The *runCycle* method of cascade composite components are very similar. The only difference is that they do not have a *changeComponentEventQueues* method as their components receive the events in the same cycle they have been dispatched.

Method *runFullCycle*, implementing the *run to completion* action, is responsible for executing the *runCycle* method as many times as it is needed to ensure that every generated event in the contained component gets processed. This is supported by the *hasInnerEvent* method that checks whether any of the contained components has unprocessed events.

Atomic components The difference between the generated class of atomic components and composite components is as follows:

- In case of atomic components the class contains a single statechart implementation object instead of the objects representing contained components.
- In case of atomic components the class contains a pair of queues responsible for storing incoming events. Both are used in case of synchronous composite components and only one for cascade composite components.

The following snippet describes these elements of the Java class generated from the gamma synchronous atomic component *TrafficLightCtrl*, presented in Section 4.3.

```
public class TrafficLightCtrl {
  // The wrapped statemachine implementation
  private TrafficLightCtrlStatemachine trafficLightCtrlStatemachine =
    new TrafficLightCtrlStatemachine();

  // Event queues for the synchronization of events
  private Queue<Event> eventQueue1 = new LinkedList<Event>();
  private Queue<Event> eventQueue2 = new LinkedList<Event>();
  ...
}
```

Synchronous component wrappers

A class representing a synchronous component wrapper has the following fields: an object representing the wrapped synchronous composite component, timer objects implementing clocks, port objects, and a multiqueue serving as a "bundle" for the separate messages queues. The multiqueue is a third-party open-source concurrent collection implementation that extends the existing Java concurrent collection library.[1] Essentially, it is a data structure with one head and multiple tails, allowing readers to block on more than one queue. It supports the definition of priorities for different sub-queues and provides round-robin selection of elements among sub-queues with the same priority. The following snippet describes the important elements of the Java class generated from the gamma synchronous component wrapper *AsyncCrossroads*, presented in Section 4.3.

```
public class AsyncCrossroads {
  // The wrapped synchronous component
  private CrossroadComponent crossroadComponent = new CrossroadComponent();
  // Timer as clock implementation
  private Timer clockSignal = new Timer();
  // Message queues
  private LinkedBlockingMultiQueue<String, Event> __asyncQueue =
    new LinkedBlockingMultiQueue<String, Event>();
  private LinkedBlockingMultiQueue<String, Event>.SubQueue executionMessages;
  private LinkedBlockingMultiQueue<String, Event>.SubQueue crossroadsMessages;

  private void init() {
    executionMessages = __asyncQueue.addSubQueue("executionMessages", 2, 8);
    crossroadsMessages = __asyncQueue.addSubQueue("crossroadsMessages", 1, 8);
    clockSignal.scheduleAtFixedRate(
      new TimerTask() {
        public void run() {executionMessages.offer(new Event("clockSignal"));}
      }, 100, 100); // Delay and period in ms
  }

  public void run() {
    while (!Thread.currentThread().isInterrupted()) {
      Event event = __asyncQueue.poll();
      if (!isControlEvent(event)) {
        // Event should be forwarded to the wrapped component
        forwardEvent(event);
      }
      performControlActions(event);
    }
  }

  private void performControlActions(Event event) {
    // Full step triggers
    if (event.getSource().equals("execution.execute")) {
      crossroadComponent.runFullCycle();
    }
    // Single step triggers
    if (event.getSource().equals("clockSignal")) {
      crossroadComponent.runCycle();
    }
    else if (event.getSource().equals("police.interrupt")) {
      crossroadComponent.runCycle();
    }
  }
}
```

[1] https://github.com/marianobarrios/linked-blocking-multi-queue

In addition to the object representing the wrapped synchronous component, a single multiqueue is defined (*___asyncQueue*) which is instantiated with two subqueues (*crossroadsMessages* and *executionMessages* in this example), representing the gamma message queues. Note that the subqueues are instantiated with parameters *priority* and *capacity* as defined in the gamma model.

As each synchronous component wrapper represents an independently running unit, the generated class implements the *Runnable* Java interface. The instances of the class can run on separate threads implementing the following behavior (see method *run*).

1. An event is retrieved from the multiqueue. The multiqueue either blocks if it is empty or returns an event from the highest priority non-empty sub-queue.
2. Method *isControlEvent* is used to check whether the event is a control event, that is, it can be processed by the wrapped component (not control event) or not (control event).
3. If the event is *not* a control event, it is forwarded to the wrapped component (*forwardEvent*) so the wrapped component can process it when a cycle is initiated.
4. The necessary control actions are performed, that is, method *performControlAction* checks whether the given event indicates the initiation of a single step (single cycle), a full step, or the reset of the wrapped component. The wrapped component is handled accordingly.

The execution of the object representing a synchronous component wrapper can be stopped by sending an interruption signal to its parent thread.

Asynchronous composite components

A class representing an asynchronous composite component has the following fields: objects representing contained asynchronous components and port objects. The generated Java class is simple: its only functionality is to bundle the contained components and forward incoming events from the system ports to the corresponding ports of contained components. This can be easily implemented in the port classes using listeners. Note that this can be considered as flattening the asynchronous hierarchy to the synchronous wrapper components, as they are the objects that store and handle asynchronous messages. Nevertheless, asynchronous composite components make a very important part in supporting *hierarchy* and the *separation of concern* and useful for the description of multi-threaded applications.

Chapter 5

Implementation

This section starts with the introduction of the various techniques and tools that have been used throughout the development of the gamma framework. Next the architecture of the framework is presented, which is followed the introduction of the modeling tools that have been integrated to the framework.

5.1 Technologies

We put a considerable amount of effort into finding the appropriate frameworks upon which the gamma framework could be implemented. As we prefer open-source technologies with receptive communities, we chose the Eclipse environment with the Eclipse Modeling Framework (EMF). Moreover, the VIATRA[1] transformation framework was used for the implementation of the model transformations and the Xtext framework for the development of the modeling language. These technologies fit well into the Eclipse environment.

5.1.1 Eclipse Environment

Eclipse[2] 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 plug-in system supports the customization of the environment for various purposes, e.g., EMF and Yakindu can be installed to support the modeling of statecharts or we can install plug-ins of our own work to support the composition and verification of reactive systems.

Eclipse Modeling Framework Eclipse Modeling Framework[3] (EMF) is an Eclipse-based modeling framework with a strong code generation support. EMF aims to facilitate the development of modeling tools and other applications offering a structured data model called Ecore. Based on the model specification defined in XMI, EMF provides runtime support and code generator tools to derive a set of Java classes describing objects of the model. Furthermore, a set of adapter classes are generated which promote users in the modification and editing of their models.

EMF is considered as a de facto standard in the development of domain-specific modeling languages, providing environment to numerous technologies and frameworks, including server solutions, persistence frameworks, UI frameworks and transformation frameworks.

[1] https://wiki.eclipse.org/VIATRA
[2] https://eclipse.org
[3] http://eclipse.org/modeling/emf/

5.1.2 Xtext Framework

Xtext[4] is an open-source Eclipse framework for the development of programming languages and domain-specific languages. 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 grammar, or can be manually given. In addition, Xtext provides several features to support development in the language: a parser, a linker, a compiler, as well as a typechecker and editing support for Eclipse (syntax highlighting, code completion, etc.).

The textual syntax of the gamma language has been built using the Xtext framework. Each part of the metamodel of the gamma language was created separately from the grammar.

Xtend[5] is a general-purpose, high-level, statically typed object-oriented programming language that is built on the Xtext framework. Xtend source code is automatically compiled to Java code, thus code written in Xtend can be integrated with all existing Java libraries easily. Also, Xtend has its roots in Java syntactically as well as semantically, however, it offers a tighter and more solid syntax. Furthermore, Xtend proposes additional functionality that is not supported by Java, e.g., type inference, operator overloading, extension methods and dispatch methods. In addition to object-oriented features, Xtend integrates traits of functional programing, such as lambda expressions, which also helps to keep the codebase small.

The model transformation and source code generation rules have been implemented using the Xtend language. Unique features, such as extension methods, dispatch methods and lambda expressions have been used extensively. As a result, the codebase has remained relatively small while the source code itself has remained readable and concise.

5.2 Architecture

The architecture of the gamma framework is plug-in-based, which makes it modular, customizable, and easily extensible. Each functionality that was introduced in Chapter 3 is implemented as a collection of Eclipse plug-ins based on EMF. Figure 5.1 depicts the architecture of the framework, presenting the plug-ins and their dependencies.

Owing to the plug-in based architecture, it is possible to use only a subset of the gamma framework functionalities by loading only the necessary plug-ins. This solution enables to save resources, e.g., reduce memory footprint.

Furthermore, the plug-in based architecture supports the easy extension of the framework. Additional engineering modeling languages as well as the analysis languages can be introduced to the gamma framework by defining the necessary model transformations and implementing them as a plug-in. These plug-ins then could be integrated into the framework easily, as they would not interfere with the existing plug-ins or their dependencies.

5.3 Integrated Modeling Languages

Currently, a single engineering and a single analysis modeling language is integrated to the gamma framework, Yakindu and UPPAAL, respectively. The integration of additional ones is supported by the plug-in architecture, as presented in Section 5.2. This section introduces

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

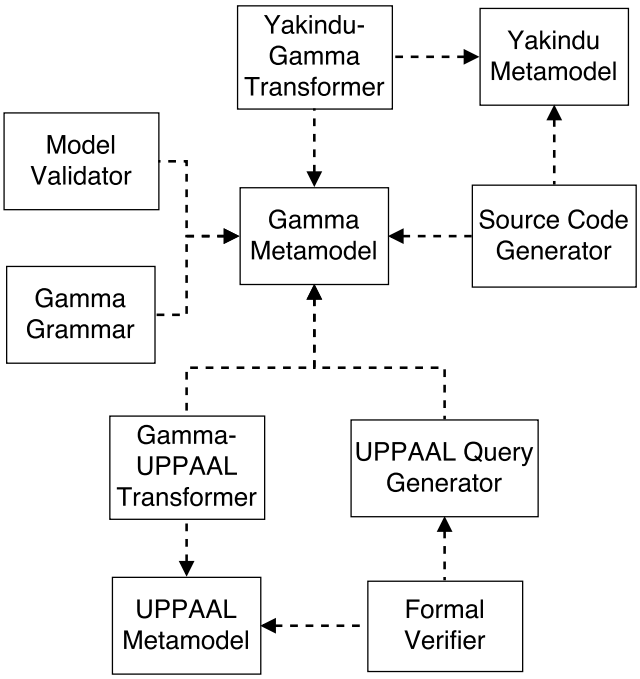


Figure 5.1: The plug-in dependencies of the framework.

the features of modeling languages Yakindu and UPPAAL, which heavily influenced the design and implementation techniques of the gamma framework.

5.3.1 Integrated Engineering Language: Yakindu

Yakindu is a toolkit for the model-driven development of reactive, event-driven embedded systems by supporting 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. A Yakindu statechart is depicted in Figure 5.2. To support users in designing well-formed statecharts the tool provides basic validation features. Although, these rules are not as comprehensive as the validation rules of gamma, live syntactic and semantic checks on the entire model are included, therefore the users get feedback on their work immediately.

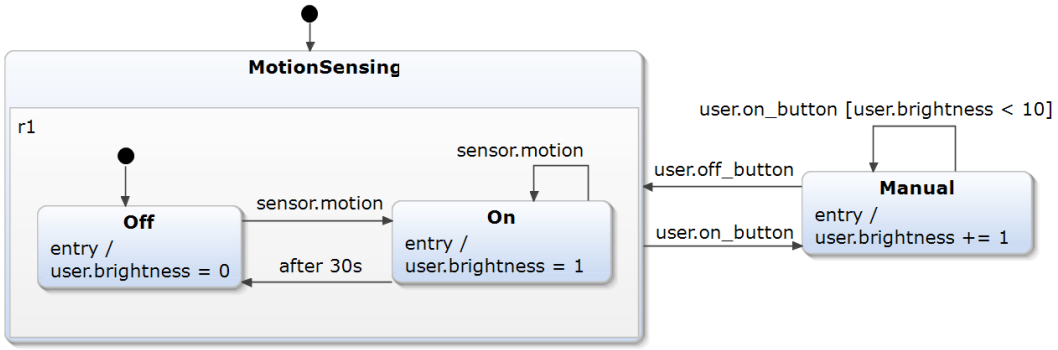


Figure 5.2: The graphical representation of a Yakindu statechart.

Syntactically correct 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, e.g., timer services and observer registration.

The gamma framework utilizes the following Yakindu functionalities:

- Gamma statecharts can be created graphically with the help of the Yakindu editor. A user can create a Yakindu statechart, which can be transformed to the gamma language by means of the Yakindu-gamma model transformer.
- The gamma framework reuses the Yakindu source code generators when generating a composite system implementation. Gamma generates only the source code that is responsible for the connection of components; the implementation of the wrapped statecharts are derived by Yakindu.

5.3.2 Integrated Model Checker: UPPAAL

UPPAAL is a software tool for the modeling, simulation, verification and validation of real-time embedded systems. UPPAAL uses the timed automata formalism which is the extension of the finite automata formalism presented in Section 2.2: it supports data types and variables as well as the *synchronization* of concurrent automata through channels. UPPAAL is capable of executing *formal verification* on the defined timed automata systems.

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 explores 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. 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.

Formal verification with the Gamma framework

As presented in Section 3.6, formal verification, back-annotation of the results and generation of a state-covering test-suite are supported with a GUI (see Figure 5.3). Thus, users do not have to deal with the generated formal models, the manual construction of CTL expressions and the handling of the UPPAAL model checker. Using this window users can formulate their conditions they want to check with regard to their constructed models.

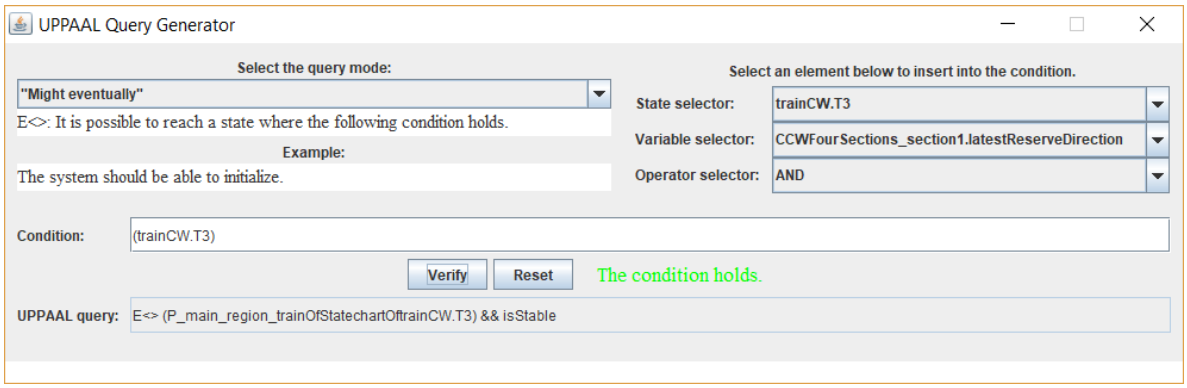


Figure 5.3: The window supporting the verification and back-annotation functionalities of the gamma framework.

On the upper-left part of the window users can choose the temporal operator that specifies the paths on which the formulated conditions must hold. Also, these temporal operators are presented with examples. The selectors on the upper-right part can be used to formulate the desired condition using **States**, **Variables** and **Operators**. Selector **States** contain the states of the model under verification, whereas selector **Variables** contain its variables. Selector **Operator** contains the operators that are accepted by UPPAAL. It is important to note that conditions can also be formulated by manually typing in the **Condition** text field.

Whether the condition holds on the model can be verified by clicking on the **Verify** button. Only well-formed conditions can be given to verification, which is checked right before starting the verification process. In cases of ill-formed conditions the user is notified in the lowermost text field.

If the given condition is well-formed, the verification starts. UPPAAL examines whether the condition holds or not and can generate a trace serving as proof or as a counterexample. Such traces are automatically back-annotated to the gamma language, so users can examine them in a familiar domain instead of the UPPAAL language. The following snippet introduces a gamma trace, which has been created as the back-annotation of an UPPAAL trace, holding information about the behavior of a synchronous composite system.

```
; trainCW.main_region_train_T1,CCWFourSections.CCWFourSections_section4
.main_Free,trainCCW.main_region_train_T1,...
<
-----
> moveTrainCCW.move,moveTrainCW.move
t 0
<
; trainCW.main_region_train_T1T2,CCWFourSections.CCWFourSections_
section4.main_Free,trainCCW.main_region_train_T1T2,...
-----
> moveTrainCCW.move,moveTrainCW.move
t 0
<
; trainCW.main_region_train_T2,CCWFourSections.CCWFourSections_
section4.main_Free,trainCCW.main_region_train_T2,...
-----
```

As mentioned in Section 4.1, synchronous composite systems adopt a turn-based semantics. A turn is called a *cycle*. These cycles are separated by the ----- delimiter in the gamma trace. At the beginning of each cycle, the composite system processes the events sent to its ports. The events and ports through which they have been received are enumerated after the “>”

mark in the form of *port_name.event_name*, e.g.,*moveTrainCCW.move*. In the first cycle there are no incoming events as the composite system must assume a stable state first. The incoming events might change the state of the composite system; the states of all instances are enumerated after the ";" mark, e.g., *trainCCW.main_region_train_T1*. A composite system might raise events as well. The events and ports through which they have been raised are after the "<" mark. In this example the composite system does not send any signals. Finally, during the execution of a cycle, time might elapse. The elapsed time in traces is represented by a non-negative integer after the *t* character. In this example, the elapse of time is not measured.

Both incoming and outgoing signals could have parameters. These parameters (their possible types are presented in Section 3.2) appear in the trace too. They are represented after the name of the port on which their signals are sent and a ":" mark, e.g. *moveTrainCCW.move:10* if the move event would have had an integer parameter of 10.

In addition to gamma traces, JUnit test classes can be generated with the GUI that are based on the UPPAAL traces. A test class based on a particular UPPAAL trace tests whether the composite system implementation, automatically generated from the particular composite model under verification, actually assumes the states that the back-annotated trace describes, i.e., in reaction to the incoming events it assumes the corresponding state and raises the necessary output events. With this technique the following functionalities of the gamma framework can be tested for a particular trace:

- Yakindu to gamma transformation,
- composite system to UPPAAL transformation,
- source code generation from the composite system,
- back-annotation of the UPPAAL trace.

As multiple functionalities can be tested using this technique, it can be considered as the validation of the gamma workflow. Naturally, the framework can be tested with as many models and traces as needed to ensure the stability of the framework.

Chapter 6

Case Study: MoDeS^3

This chapter demonstrates the applicability of the gamma framework by presenting a case-study from the domain of critical cyber-physical systems. The selected case-study is the so-called MoDeS^3, which is based on a railway transportation system. The trains are controlled by the users, but a distributed safety logic is responsible for preventing dangerous situations, such as the collision of trains.[1]

6.1 Introduction

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. MoDeS^3 is a physical railway model. Multiple trains move on tracks, which are built from sections and turnouts. The trains can be controlled by users by changing their movement direction and speed. Turnouts are also controlled by users: their directions 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 is responsible for gathering all the information that these components 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, therefore, a good case study for gamma.

6.2 Interlocking Safety Logic

Safety has to be assured inside a single zone, as well as on the edges of zones. Since embedded computers (BBBs) have only local information about their own zone, MoDeS^3 can be considered as a distributed system. This makes accident prevention difficult, since information has to be gathered in one zone, which then has to be delivered to controllers of adjacent zones via a network. As incidental packet losses might have critical consequences, the distribution of information has to be supported by a reliable protocol.

The safety system responsible for preventing the collision of trains is based on statecharts. Two different statecharts have been designed, one to describe the behavior of a single section, and

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

another one to describe the behavior of a single turnout. These statecharts have been designed in Yakindu and transformed to the gamma language. A single statechart is associated to each element of a zone corresponding to its type. The statecharts of a zone have been composed using the gamma framework, thus a composite component model is created, which describes the behavior of an entire zone.

6.2.1 Interfaces

As we have multiple statechart models that need to communicate with each other, it is important to define the interfaces through which communication will take place. The interfaces that ports of the gamma statecharts realize are the following.

```
interface Protocol {
    in event canGo
    in event cannotGo
    in event reserve
    in event release
}
interface Control {
    in event restartProtocol
}
interface Turnout {
    in event turnoutStraight
    in event turnoutDivergent
}
```

```
interface Train {
    in event stop
    out event occupy
    out event unoccupy
}
interface MoveTrain {
    in event moveForward
    in event moveBackward
}
```

Interface *Protocol* contains events that are used in section-to-section and turnout-to-section communication. The semantics of the *Protocol* events are as follows.

- Reserve: this event is sent to adjacent sections from a section occupied by a train.
- CanGo: this event 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 event 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 event is sent to adjacent sections from a section just left by a train.

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

- Occupy: this event is sent to a section if it has been occupied by a train.
- Unoccupy: this event is sent to a section if a train has left it.
- Stop: this event is sent to the corresponding controller if the section wants to stop the train standing on it.
- RestartProtocol: this event is used for resetting sections in state Stop.

Interface *Turnout* contains events which can be sent to turnouts to change their directions. Interface *MoveTrain* contains events that can be used for the controlling of a model train, used in the verification process.

6.2.2 Section Statechart

The section statechart has been abstracted to focus only on the qualities of the real sections that are relevant in the safety logic. As a section has two endpoints, the model concentrates on two directions it can receive or send protocol events to; these directions are called clockwise (CW) and counterclockwise (CCW). Communication with other section and turnout statecharts is supported with the use of ports; the section statechart has two ports for both directions, one for realizing interface Protocol in required mode and another one for provided mode. The communication with the corresponding controller is also supported by ports realizing interface *Control* and *Train*.

```
port protocolProvidedCW : provides Protocol
port protocolProvidedCCW  : provides Protocol
port protocolRequiredCW : requires Protocol
port protocolRequiredCCW : requires Protocol

port controlProvided : provides Control

port trainRequired : requires Train
```

The behavior of the section statechart is going to be introduced using a track segment model consisting of four sections, which is depicted in Figure 6.1.

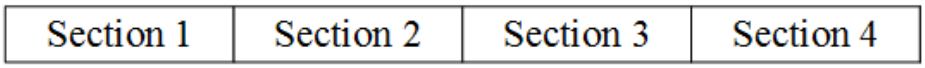


Figure 6.1: A track segment containing four consecutive sections.

Each section not affected by any train is in state *Free*. If a train is placed onto *Section 2*, it goes to state *Occupied*. The adjacent sections *Section 1* and *Section 3* go to state *Reserved*. The sections in state *Reserved* and in state *Occupied* (*Section 1*, *Section 2* and *Section 3*) are said to be in the "aura" of the particular train. If the train moves from *Section 2* and reaches adjacent *Section 3*, *Section 3* tries to reserve the other adjacent section *Section 4*. If the reservation is successful, *Section 3* and *Section 4* go to states *Occupied* and *Reserved*, respectively. It is important to note that section *Section 2* stays in state *Occupied*. When section *Section 2* is left entirely by the train, it goes to state *Reserved*, while releasing *Section 1*, which goes to state *Free*.

In the previous example *Section 3* successfully reserved *Section 4*. On the other hand, if the reservation had failed, *Section 3* would have gone to state *Stop*. Generally, if a section in state *Reserved* gets any more reservation claims by sections not belonging to its own "aura", it will respond with a *cannotGo* event. 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. Sections in state *Stop* also send *cannotGo* events in response to incoming reservation claims, which ensures to stop both of the trains proceeding towards one other.

Figure 6.2 demonstrates the behavior of section *Section 3* when it is reached by the train.

Generally, 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 following the other one is stopped; the train on the front may continue its way.

Sections in state *Stop* can be reset in one of the following ways. A *restartProtocol* event 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*

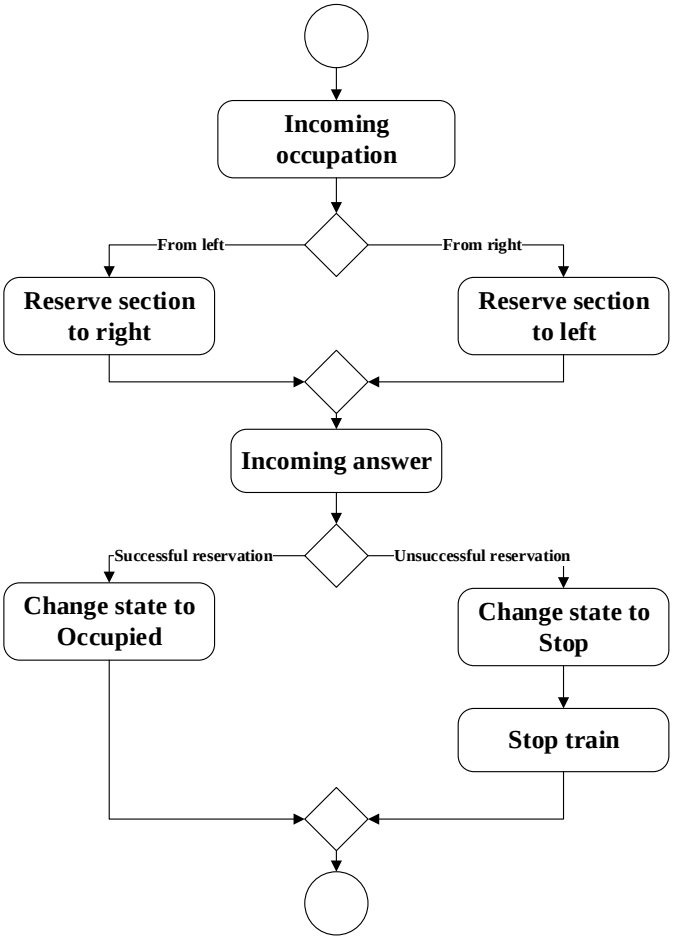


Figure 6.2: The behavior of a section statechart upon receiving an occupation event.

event to the section in state *Stop*, on which it goes into state *Free* while sending *Release* events to adjacent sections.

6.2.3 Turnout Statechart

The turnout statechart has been designed similarly to the section statechart. It supports each event used in section-to-section and section-to-train communication. Furthermore, it supports events *turnoutStraight* and *turnoutDivergent*. Since trains must not stop on turnouts, events *stop* and *restartProtocol* are not supported by turnout statecharts.

A turnout has three sides, each connected to a single section. The sides, and thus the sections from the turnout point of view are named as *top*, *straight* and *divergent* (see Figure 6.3).

Figure 6.3: Naming convention of sections from a turnout's point of view.

A turnout has two states. In state *Straight* it connects sections top and straight, in state *Divergent* it connects sections top and divergent.

Events of section-to-turnout 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 state *Straight* or *Divergent*. The adjacent section processes the event and if it has any responses, they are transmitted back to the section initiating the event exchange.

Events *turnoutStraight* and *turnoutDivergent* 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 when a user changes the state of a turnout while a train proceeds towards its *straight* or *divergent* endpoint must be addressed. In these situations, the endpoint gets locked-up, the turnout state is switched (*Straight* to *Divergent* or vice versa), and the section under the train is put into state *Stop*.

6.3 Supporting the Development and 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. Many flaws 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 (see Section 6.4).

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 composition language. The corresponding ports of the statechart components were connected, which was followed by the generation of source code. Due to lack of time, we did not manage to deploy the source code onto the BBB controllers, but we are going to work on it in the near future.

6.4 Formal Verification of the Safety Logic

Analysis of the safety logic can focus on *1)* the interaction of elements in a single zone as well as *2)* the interaction of multiple BBBs. The latter one is based on the implemented network protocol, therefore this work presents the former one. The presented model is a *synchronous composite component* model since the interleaving of possible events in case of asynchronous composite models would cause a state space explosion, making it impossible for UPPAAL to evaluate any non-trivial requirement on the generated automata.

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 have to be detected by the same safety logic.

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 *moveForward* or *moveBackward* event of the port realizing interface *MoveTrain* in provided mode, which symbolizes the proceeding of the train. Also, there is a boolean variable *disabled* which can be set to prevent further movement.

A hierarchical composition model has been created to model the railway system. The track model under verification is going to consist of a sequence of six sections, which can be constructed using two instances of a track model containing a sequence of three sections. The track model with three sections is as follows.

```
sync ThreeSections [
    // Ports for events on the sides
    port protocolProvidedCCW : provides Protocol
    port protocolRequiredCCW : requires Protocol
    port protocolProvidedCW : provides Protocol
    port protocolRequiredCW : requires Protocol
    // Ports for controlling the trains
    port trainRequired1 : requires Train
    port trainRequired2 : requires Train
    port trainRequired3 : requires Train
] {
    // Section instances
    component section1 : Section
    component section2 : Section
    component section3 : Section
    // Binding system ports to the ports of component instances
    bind protocolProvidedCCW -> section1.protocolProvidedCCW
    bind protocolRequiredCCW -> section1.protocolRequiredCCW
    bind protocolProvidedCW -> section3.protocolProvidedCW
    bind protocolRequiredCW -> section3.protocolRequiredCW
    bind trainRequired1 -> section1.trainRequired
    bind trainRequired2 -> section2.trainRequired
    bind trainRequired3 -> section3.trainRequired
    // Connecting ports of adjacent sections
    channel [section1.protocolProvidedCW] -o)- [section2.protocolRequiredCCW]
    channel [section2.protocolProvidedCCW] -o)- [section1.protocolRequiredCW]
    channel [section2.protocolProvidedCW] -o)- [section3.protocolRequiredCCW]
    channel [section3.protocolProvidedCCW] -o)- [section2.protocolRequiredCW]
}
```

The track model consisting of six sections can be constructed as follows:

```
sync SixSections [
    // Ports for controlling the trains
    port moveTrainCCW : provides MoveTrain
    port moveTrainCW : provides MoveTrain
] {
    // Instantiating two trains and two times three sections
    component threeSectionsCCW : ThreeSections
    component threeSectionsCW  : ThreeSections
    component trainCCW : Train
    component trainCW : Train
    // Binding system ports to the ports of train component
    bind moveTrainCCW -> trainCCW.Train
    bind moveTrainCW -> trainCW.Train
    // Connecting the track segments
    channel [threeSectionsCCW.ProtocolProvidedCW] -o)-
      [threeSectionsCW.ProtocolRequiredCCW]
    channel [threeSectionsCW.ProtocolProvidedCCW] -o)-
      [threeSectionsCCW.ProtocolRequiredCW]
    // Connecting the trains to sections ...
}
```

As can be seen, the synchronous composite component *SixSections* consists of six consecutive sections. In this example they are going to be called *section1*, *section2*, $\cdots$, *section6*, where *section1* is component *section1* of *threeSectionsCCW* and *section6* is component *section3* of *threeSectionsCW*. 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* event 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 (events *occupy* and *unoccupy*) and sections are enabled to stop trains (event *stop*). This can be done in two separate modes each modeling one of the dangerous situations:

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 6.4. The corresponding channel definition in synchronous composite component model *SixSections* is as follows:

```
channel [trainCCW.TrainProvided1] -o)- [threeSectionsCCW.TrainRequired2]
channel [trainCCW.TrainProvided2] -o)- [threeSectionsCCW.TrainRequired3]
channel [trainCW.TrainProvided1] -o)- [threeSectionsCW.TrainRequired2]
channel [trainCW.TrainProvided2] -o)- [threeSectionsCW.TrainRequired1]
```

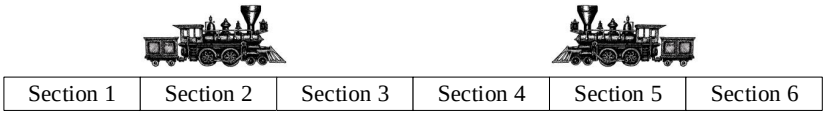


Figure 6.4: Layout: two trains proceed towards each other.

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 6.5. The corresponding channel definition in synchronous composite component model *SixSections* is as follows:

```
channel [trainCCW.TrainProvided1] -o)- [threeSectionsCCW.TrainRequired2]
channel [trainCCW.TrainProvided2] -o)- [threeSectionsCCW.TrainRequired3]
channel [trainCW.TrainProvided1] -o)- [threeSectionsCW.TrainRequired2]
channel [trainCW.TrainProvided2] -o)- [threeSectionsCW.TrainRequired3]
```

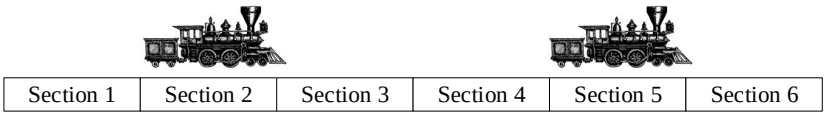


Figure 6.5: Layout: a train proceeds to another one from behind.

Two separate composite models have to be constructed in these ways, so both dangerous situations can be analyzed. With the use of the gamma-UPPAAL model transformer the composite model can be transformed to UPPAAL and verification can begin.

6.4.1 Analysis of the First Layout

As a result of the gamma-UPPAAL model transformation, eight automata are 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. UPPAAL has shown that the requirements are satisfied by the designed composite system. This proves the following statements:

1. There is no deadlock in this layout, i.e., the UPPAAL models are valid.
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.

6.4.2 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`, that is, the system is deadlock free.
- `A[] !(Process_main_region_trainOfStatechartOfTrain1.T2 && Process_main_region_trainOfStatechartOfTrain2.T1)`, that is, train1 is never positioned on section3 completely while train2 is facing in the other direction and is positioned on section4 completely.
- `A[] !(disabledOfTrain2)`, that is, train2 can never be stopped by the safety logic in this layout .

UPPAAL has shown that the requirements are satisfied by the designed composition system. This proves the followings:

1. There is no deadlock in this layout, i.e., the UPPAAL models are valid.
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.

6.5 Summary

In summary, the correctness of the MoDeS^3 safety logic designs has been proven with the help of the gamma framework. As a result, the source code generated from the composition models will also work correctly in these situations owing to the formal composition semantics both UPPAAL models and the generated source conform to.

Chapter 7

Conclusion

In this work we presented the gamma framework, which supports the design, implementation and verification of state-based reactive sytems using model-driven software development concepts. The core of gamma is its statechart language, which is supported by the Yakindu Statechart Tools for high-level design, a Java code generator for implementation and the UPPAAL model checker for formal verification. Moreover, its extensible architecture allows additional tools and features to be plugged in.

The main contribution of this work is the extension of the gamma framework with features for hierarchical composite modeling. First of all, we extended the statechart language with elements for composition. These new elements define ports and interfaces, enabling individual components to serve as endpoints. Communication is provided by channels connecting port instances. Relying on these elements, we defined various kinds of components for hierarchical composite model building. The three distinguished composition modes are the asynchronous-reactive, the synchronous-reactive and cascade.

Asynchronous components represent independently running components, which communicate with immutable messages stored in message queues. This semantics is suitable for designing separate units executed in their own processes. Synchronous-reactive components are useful for providing a single executing unit consisting of multiple, functionally independent components. This composition mode is beneficial for the design of low-level controllers. Cascade composition is practical for designing units with pipeline-like behavior: the input given into the model is processed by multiple consecutive filters. We believe that these composition methods cover a large portion of the problems emerging in the design of reactive systems.

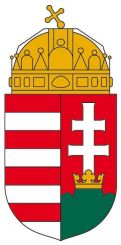
From the theoretical point of view, we also defined the precise semantics of the aforementioned composition modes. This enabled us to extend the currently existing code generation and verification tools in the gamma framework to be able to handle hierarchical composite models. These functionalities are not yet complete, as for example code generation and model checking currently support only synchronous models. However, these deficiencies can be completed rather easily as we already have detailed plans on how they should work.

Furthermore, we plan to integrate ongoing side-projects, aiming to extend the gamma framework with additional functionalities, including source code generation from gamma statecharts and code generation to distributed controllers with network communication. Moreover, we also plan to extend the framework with additional engineering tools, e.g., MagicDraw and Stateflow, and analysis modeling tools, e.g., Spin [5].

By offering multiple modeling aspects, compositional semantics, source code generation and verification functionalities in a single, extensible framework, we hope that gamma can assist system and software engineers in leveraging the potential of model-driven development.

Acknowledgements

First and foremost, I would like to thank my advisor, Vince Molnár for his guidance and continuous support. Furthermore, I would like to thank Ákos Hajdu for his valuable reviews and ideas. I am also grateful to Dr. István Majzik and András Vörös for their valuable comments.



EMBERI ERŐFORRÁSOK
MINISZTÉRIUMA

Supported by the ÚNKP-17-2-I New National Excellence Program of the Ministry of Human Capacities.

Bibliography

[1] *The Mathworks: Stateflow and Stateflow Coder, User's Guide*. 2003.

[2] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Rigorous implementation of real-time systems - from theory to application. *Mathematical Structures in Computer Science*, 23(4):882–914, 2013.

[3] 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.

[4] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. Ieee, 2006.

[5] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer London, 2008.

[6] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Software and Systems Modeling*, April 2014.

[7] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. Component-based verification using incremental design and invariants. *Software and System Modeling*, 15(2):427–451, 2016.

[8] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[9] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in bip. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 77–86. ACM, 2009.

[10] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 3(1):1–207, 2017.

[11] Christopher Brooks. Ptolemy II: An open-source platform for experimenting with actor-oriented design, February 2016. Poster presented at the "https://www.terraswarm.org/conferences/16/bears/" 2016 Berkeley EECS Annual Research Symposium.

[12] Richard Colgren. *Basic Matlab, Simulink And Stateflow*. AIAA (American Institute of Aeronautics & Ast, 2006.

[13] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21 – 42, 2003.

[14] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[15] Ylies Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Software and System Modeling*, 14(1):173–199, 2015.

[16] Bence Graics. Model-Driven Design and Verification of Component-Based Reactive Systems. Students's Association Report, Budapest University of Technology and Economics, 2016.

[17] Bence Graics. Model-Driven Design and Verification of Component-Based Reactive Systems. Bachelor's thesis, Budapest University of Technology and Economics, 2016.

[18] Bence Graics and Vince Molnár. *Formal Compositional Semantics for Yakindu Statecharts*, page 22–25. Budapest University of Technology and Economics, Department of Measurement and Information Systems, Jan 2017.

[19] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[20] Grégoire Hamon. A denotational semantics for stateflow. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172. ACM, 2005.

[21] Grégoire Hamon and John Rushby. An operational semantics for stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5):447–456, Oct 2007.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[23] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[24] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. 1976.

[25] Steven T Karris. *Introduction to Stateflow with Applications*. Orchard Publications, 2007.

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

[27] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[28] Edward A. Lee. Finite state machines and modal models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[29] Edward A Lee and Eleftherios Matsikoudis. The semantics of dataflow with firing. *G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 71–94, 2008.

[30] Edward A. Lee and Haiyang Zheng. *Operational Semantics of Hybrid Systems*, pages 25–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[31] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 114–123, New York, NY, USA, 2007. ACM.

[32] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

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

[34] Anis Sahbani and Jean-Claude Pascal. Simulation of hyibrd systems using stateflow. In *ESM*, pages 271–275, 2000.

[35] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.

[36] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. `http://www.csl.sri.com/~tiwari/-stateflow.html`.

[37] Sergio Yovine. BIP: Language and tools for component-based construction, 2007.