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

Mixed-Semantics Composition of Statecharts for the Model-Driven Design of Reactive Systems

Master's Thesis

Author
Bence Graics

Advisor
Vince Molnár

December 7, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Graics Bence*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. december 7.

Graics Bence
hallgató

Kivonat

Az elmúlt évtizedek technológiai fejlődésének eredményeképp napjainkban rendkívül széles körben alkalmaznak programozható vezérlőket akár olyan kritikus területeken is, mint például az autóipari beágyazott rendszerek. A „intelligens” eszközök korában ráadásul több szintre váltak szét az egy feladatot ellátó programok – például egy jármű berendezéseit közvetlenül irányító beágyazott számítógépeken futó alkalmazások szoros kapcsolatban állnak a központi számítógéppel, az pedig a felhőben futó szolgáltatásokkal. Az ilyen heterogén rendszerek komplexitása nagyon nagy lehet. A komplexitás kezelésének gyakori módszere a modellvezérelt fejlesztés, amely lehetővé teszi, hogy a fejlesztő a technikai részletek helyett a probléma logikai aspektusaira koncentráljon.

Jelen dolgozatban egy olyan, modellezést támogató eszközt mutatunk be, amely a fent körvonalazott rendszerek alábbi sajátosságaira ad választ.

1. Komponens alapú felépítés: A megcélzott rendszerek jellegzetessége, hogy komponensekből épülnek fel – ezért egy hierarchiát támogató modellezési nyelvre van szükség.

2. Kommunikáció: A komponensek jellegzetesen logikai jelekkel vagy üzenetekkel kommunikálnak – ennek támogatására jól definiált interfészekre és kommunikációs pontokra van szükség.

3. Elosztott működés: A komponensek nem csak egyetlen szoftvert reprezentálhatnak, hanem több, különböző fizikai eszközön futó programot alkothatnak – ez heterogén kommunikációt jelent, amihez többféle kompozíciós szemantikára van szükség.

4. Minőségi és helyességi követelmények: A rendszerek (bizonyos részei) gyakran kritikus feladatokat látnak el, ahol a helyes működés alapvető elvárás – ehhez helyes modellek építésére van szükség, ami validációval, verifikációval, a minőségi implementáció pedig automatikus kódgenerálással és teszteléssel támogatható.

A jelen dolgozatban bemutatott Gamma Állapotgép Kompozíciós Keretrendszer egy hierarchikus, komponens-alapú reaktív rendszerek modellezését támogató eszköz. Az elemi komponensek viselkedésének leírásához az eszköz saját formális nyelvén túl a keretrendszerbe illeszthető külső eszközök (pl. Yakindu Statechart Tools) is használhatók. A komponensek háromféle szemantika szerint komponálhatók: az aszinkron-reaktív szemantika az elosztott kommunikáció modellezéséhez, a szinkron-reaktív szemantika az egy programon belüli vagy nagymértékben szinkron kommunikációhoz, a kaszkád kompozíció pedig egyetlen funkció logikai dekomponálásához nyújt megfelelő absztrakciót. A modellezési folyamatot a keretrendszer validációs szabályokkal segíti mind komponens, mind rendszer szinten. Az elkészített modellek minőségének biztosításához a keretrendszerhez illeszthető, a felhasználók elől elrejtett modellellenőrző eszközök (pl. UPPAAL) használhatók. A modellek alapján az implementáció automatikusan származtatható, amelynek helyessége a szintén automatikusan generálható validációs tesztkészlettel ellenőrizhető.

A keretrendszer kiterjedt funkcionalitását és az általa nyújtott lehetőségeket egy vasúti témájú esettanulmánnyal szemléltetjük.

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 central 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.

This work presents 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 composition 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 Statechart Composition 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 demonstrated through a railway-themed case study.

Chapter 1

Introduction

Statecharts [1] are an expressive language to model the behavior of reactive systems, which process events coming from the environment and react to them in accordance with their internal states. Statecharts offer a powerful formalism to describe dynamic aspects of system behavior by introducing complex constructions: composite states for hierarchical state refinement, parallel regions for describing parallel behavior, history states and variables for expressing memory, and choices, fork and join transitions to model complex transitions and actions.

Modeling standards like UML and SysML have adopted the statechart formalism, but, they did not specify its dynamic semantics either formally or algorithmically [2, 3]. As a result, design tools, such as Rhapsody[1], BridgePoint[2], MagicDraw[3] and Yakindu Statechart Tools[4] support slightly different variants of the formalism [4, 5].

System integrators, such as car manufacturers and airframers rely upon refined modeling and verification tools and use component- or platform-based design techniques to mitigate the increasing complexity of their systems [6] . In such cases, the system components can be provided by multiple vendors (subcontractors) that may use different modeling tools and even different statechart formalisms in their development process. Therefore, the composition of individual components in a semantically well-founded way is a significant challenge for system integrators.

In modeling standards the functional decomposition of components is supported on a rather syntactic level: the Component Diagram of UML and the Internal Block Diagram of SysML provide support to capture what type of information (data or control) can be sent between components, but they do not detect behavioral inadequacies in the communication. This problem is mainly caused by the lack of formal semantics regarding the behavioral aspects in those languages.

The Gamma Statechart Composition Framework is an integrated modeling tool that aims to support the semantically well-founded composition of heterogeneous statechart components where individual components may use different statechart semantics possibly coming from different modeling tools. The Gamma framework intentionally reuses statechart models of existing tools and their respective code generators for individual components. Furthermore, it provides the Gamma Composition Language that supports the interconnection of components in a hierarchical way by mixing various composition semantics

[1] `https://www.ibm.com/us-en/marketplace/rational-rhapsody`
[2] `https://xtuml.org/`
[3] `https://www.nomagic.com/products/magicdraw`
[4] `https://www.itemis.com/en/yakindu/state-machine/`

(e.g., asynchronous-reactive and synchronous-reactive). Additionally, Gamma provides automated code generators and test case generators for the interaction between components. Gamma also supports system-level formal verification and validation (V&V) by mapping statechart and composition models into formal models in UPPAAL.

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 part of the framework's core.

The first prototype of the Gamma framework The semantical inconsistencies of Yakindu and the need for integrating code from multiple statechart models (e.g., in the MoDeS^3 project[5]) led to the design of the first version of the Gamma framework. The goal was to support 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 [7].

Formalization of the synchronous compositional semantics The previously implicitly defined semantics of the composition (defined by means of model transformations) was formally defined in a conference paper [8].

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 in another composite system. Along with this improvement, the concept of ports and interfaces were introduced to define the "signature" of components and group related events into well-defined points of service. A part of this work has been presented in the Bachelor's Thesis of the author [9], 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 were presented in a tool paper on the 40th International Conference on Software Engineering, also marking the public debut of Gamma 1.0 [10]. This version of the framework was made available along with a tutorial.[6]

[5] `https://inf.mit.bme.hu/research/projects/modes3`
[6] `http://gamma.inf.mit.bme.hu/`

Gamma 2.0 In the *current phase of the project*, our goal is to broaden the modeling power of the composition language by introducing new composition modes: 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. Furthermore, this language serves 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. As the result of this work, Gamma 2.0 has been finished and released to the public as an open-source project.[7]

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 based on the DDS standard[8]), 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 In the next version of the framework we plan to develop methods for the verification of dynamic architectures, e.g., cyber-physical systems, by supporting dynamic, contract-based definition of components and connections. The verification of such models could be based on both development-time and runtime verification methods. Furthermore, we plan to tighten the integration to the upcoming versions of Yakindu. Finally, we plan to improve usability of the Gamma framework by introducing 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 as well as related modeling tools. Chapter 3 describes all features and functionalities of the Gamma framework, providing an insight to the reader on how the current work is connected to previous work. The main contribution of the current work, that is, the extension of the Gamma language family, is presented in Chapter 4. Chapter 5 presents the architecture of the Gamma framework as well as the employed technologies and integration of third-party modeling languages and the Java platform via a code generator. 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.

[7] `https://github.com/FTSRG/gamma`
[8] `https://www.omg.org/spec/DDS/1.4`

Chapter 2

Background

This chapter introduces the concepts 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 behavioral models. Next, we introduce the concept of composite reactive modeling, which is the main motivation and 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 place of the original element with respect to a certain purpose [11].

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 [12]. 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 [11].

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 systems recommending modeling techniques and technologies, such as *model-driven architecture* [13] and *model-centric software development* [14].

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)[2]. 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 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.

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

[2] Generally, programming languages can be considered as a special type of modeling languages.

State machine is a mathematical model of computation to describe the behavior of a system, component or object in an event-driven way [15]. 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.
- 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 $I \cap O = \emptyset$.
- $T \colon (I \times S) \to (S \times O)$ is the fully defined transition function that represents changes of states in response to input events and generating output events meanwhile.

The behavior of a state machine can be described with a trace. A trace consists of a sequence of steps. A step describes a change of state with an output event raising in the state machine in response to an input event. Formally, a trace of a state machine is described as follows:

- $\rho = ((i_1, s_1, s'_1, o_1), \ldots, (i_n, s_n, s'_n, o_n))$ is a trace, which consists of one or more steps: $n \in \mathbb{N}, 1 <= n$.
- (i_i, s_i, s'_i, o_i) is a step, consisting of an input event $i_i \in I$, a source state $s_i \in S$, a target state $s'_i \in S$ and an output event $o_i \in O$. A step is considered valid if $T(i, s_i) = (s_o, o)$.
- A trace $\rho = ((i_1, s_1, s'_1, o_1), \ldots, (i_n, s_n, s'_n, o_n))$ is considered valid if:
  - $s_1 = s^0$, that is, the source state of the first step is the initial state;
  - For each step (i_i, s_i, s'_i, o_i), $s_i = s'_{i-1}$, that is, the source state of a particular step is the target state of the previous step.

There are various extensions to the state machine formalism that facilitate the compact modeling of hierarchical and concurrent systems [16]. The most relevant one to this work is statecharts [1], 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.

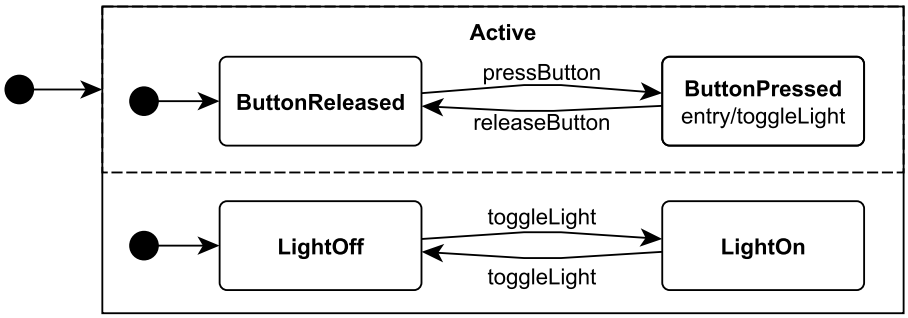

Figure 2.1: The graphical representation of the most important elements of statecharts.

A statechart contains a single region, called top region. 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. States may execute certain actions upon entering or leaving the particular state. Composite states can contain one or more regions that contain additional states. If multiple regions are contained by a particular state, they are called orthogonal. 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 Formal Verification and Model Checking

Formal verification is a method for proving or disproving the correctness of a system with mathematical precision, where the correctness is checked with respect to certain properties or specifications given by the user. *Model checking* [17] is a formal verification technique, which explores the behavior of the given model automatically and often exhaustively, i.e., all relevant behaviors of the model are analyzed, contrary to simulation and testing, which only sample behaviors. Usually, these models are represented as finite-state systems. Typically, model checking is used in the design of safety-critical systems, since their specifications often consists of safety-requirements that have to be met by all means, for example the absence of deadlocks and the unreachability of error states that may cause a system malfunction.

To carry out model checking, both the model of the system and the requirements have to be created in a formal mathematical language (see Figure 2.2). The problem, which has to be solved by the model-checker, is formulated as a task in mathematical logic, namely to check whether the given system model satisfies a given logical formula. This general concept applies to many types of mathematical logics (e.g., linear temporal logic and computation tree logic) and formalisms (e.g., Kripke-structures, labeled transition systems).

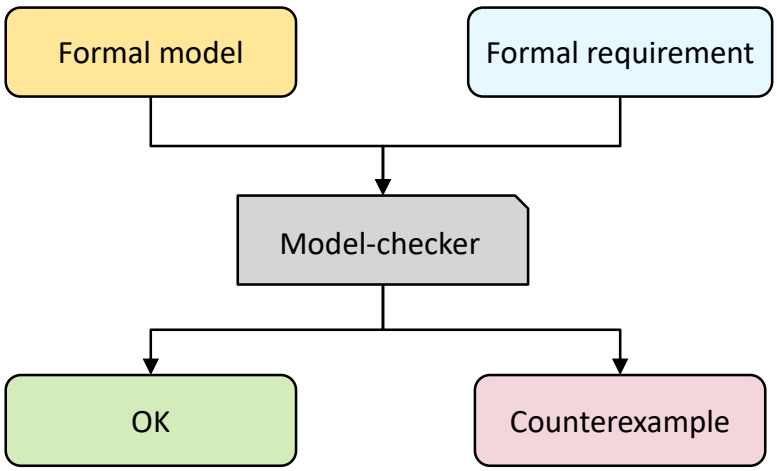

Figure 2.2: The schematic description of the model checking process.

During the model checking process, the model-checker explores the relevant part of the state space of the given formal model with regard to the formal requirement. Beside the result, the model checker may return a witness proving the correctness or incorrectness of the model.

2.4 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. A *semantic domain* is the implementation of a certain model of computation [18].

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 static analysis of deadlock-freedom and boundedness. Schedulings of components can also be computed statically [19]. 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 with Petri nets [20].

Asynchronous-Reactive In the asynchronous-reactive model, system components represent concurrent processes that communicate with each other using message queues [21]. Writing to the message queues always succeeds instantly, whereas reading from an empty queue blocks the reader process (nonblocking-write, blocking-read approach). 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 [22].

Synchronous-Reactive The synchronous-reactive model has a notion of time and follows the semantics of synchronous programming languages [23, 24, 25]. 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

	Native statecharts	Mix-and-match composition	Formal semantics	Synchronization-based	Event-based	Stochastic proc. alg.-based	Formal verification	Model-based test generation	Code generation	Simulation
Gamma	✓	✓	✓		✓		✓	✓	✓	
SystemC ME			✓		✓		✓		✓	✓
AEmilia			✓			✓	✓			✓
Ptolemy II		✓	✓	✓	✓		✓̸		✓̸	✓
BIP			✓	✓			✓		✓	✓
RSARTE MC	✓		✓		✓		✓		✓	✓

Table 2.1: Features of Gamma and its competitors. ✓= full support; ✓̸= experimental

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 asynchronous-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 can be considered as the generalization of the *synchronous circuit* model widely used in the field of digital techniques. It 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 Programmable Logic Controller (PLC) programs. 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.

2.5 Related Work

As related work, we cover in detail the tools that simultaneously provide *1)* a composition language for component-based design with *2)* precise formal semantics and *3)* formal verification support for behavioral properties. As such, we exclude approaches such as [26, 27, 28, 29, 30, 31] that miss at least one of these aspects. The feature comparison of the following tools can be found in Table 2.1: a SystemC modeling environment (SystemC ME) connected to the STATE tool, AEmilia ADL/TwoTowers, BIP, Ptolemy II, and a model checker tool for RSARTE models (RSARTE MC).

SystemC ME In [32] a modeling environment is introduced that supports the graphical definition of SystemC [33] models. SystemC is a C++ library offering classes and macros,

which provide an event-driven simulation interface suitable for simulating concurrent processes. The basic building block of a SystemC model is the module, which represents computational parts of the design. Modules are composed of processes, ports, events, channels and variables. Processes are the main computation elements of the module, they are concurrent and are used to describe functionality. A state machine formalism can be used to define the behavior of a process. Ports allow communication from inside a module to the outside. They use interfaces to describe the type of events they are capable of handling. Ports are connected by channels. The modeling environment supports the automatic SystemC code generation from the created models. The supported part of the SystemC language is given a formal semantics by connecting the modeling environment to the STATE tool [34]. STATE maps the informal SystemC code to a formal timed automaton formalism, UPPPAL, thus provides formal verification capabilities. Contrary to Gamma, this modeling environment currently does not support the hierarchical composition of modules.

AEmilia ADL/TwoTowers AEmilia [35] is an architecture description language (ADL) based on EMPA$_{gr}$ process algebra, a compositional specification language of algebraic nature integrating process algebra theory and stochastic processes. This language supports the modeling of component-based software systems at the software architecture level. Designers have to start the modeling process by defining the behavior of the component types in the system and their interactions with the other components. Stochastic aspects, e.g., component interaction time, of the software architecture targeted for functional or extra-functional analysis (security and performance) have to be defined on this level. Next, instances of component types have to be defined along with their interactions in order to enable the communication of instances. Finally, based on the received composite models integrated, functional and performance semantic models can be generated automatically, which can undergo formal analysis executed by the TwoTowers tool. The biggest difference between the AEmilia ADL and the Gamma Composition Language is that Gamma puts the focus on statecharts instead of stochastic process algebra.

BIP BIP [36] (Behavior, Interaction, Priority) is a modeling framework supporting the formal definition of heterogeneous systems. The BIP language supports the layered definition of hierarchical composite systems, defining three layers. The lowest layer specifies the behavior of system components, atomic or compound, using a variant of the Petri net formalism. The intermediate layer consists of a set of connectors linking ports together, thus defining the interactions between transitions of components. Contrary to Gamma, these interactions are based on synchronization. The top layer includes a set of dynamic priority rules between interactions and can be used for the specification of scheduling policies. BIP defines a clear operational semantics, which describes the behavior of both atomic and compound components. The behavior of atomic components are based on a rigorous transition system model, thus, formal verification of invariant properties and deadlock-freedom is also supported.

Ptolemy II Ptolemy II [19] is a modeling framework supporting the definition of hierarchical composite systems with diverse component types and interaction semantics. Modeling components, called actors in Ptolemy II, can be regarded as independent software modules. They are able to interact with each other by sending messages through interconnected ports. Models are created by composing actors, which is supported on multiple hierarchy levels. The interactions of actors 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, e.g., process network, synchronous reactive and synchronous dataflow. The implementation of a MoC is called director. Each level of hierarchy in a model must have a single director that specifies the MoC. 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. Nevertheless, Ptolemy II offers only experimental formal verification capabilities [37].

RSARTE MC In [38] a model checking tool is presented that is capable of verifying UML-RT [39] models, which is an extension of UML supported by the RSARTE[3] modeling framework. The basic building block of an UML-RT model is a *capsule*, whose behavior can be defined using statecharts. Additionally, UML-RT models contain a structure diagram for each capsule, which describes relationships with other capsules. A capsule can contain *parts*, which are instances of other capsules, thus hierarchical modeling is supported. Capsules and parts communicate through ports. Such UML-RT models are transformed to a rigorous state machine formalism, called CFFSM [40], which can be formally analyzed by their model checker against CTL expressions.

[3] `www.ibm.com/support/knowledgecenter/SS5JSH/rsart_family_welcome.html`

Chapter 3

The Gamma Framework Features

This chapter introduces the features and functionalities of the Gamma framework that support the model-based design, implementation, validation and verification of component-based reactive systems.

3.1 Overview

Figure 3.1 depicts the functionalities of the Gamma framework. Every functionality relies on the Gamma Statechart Language (introduced in Section 4.5), which provides a common basis regarding model elements and behavioral semantics. Statecharts created with the Gamma Statechart Language can be composed using the Gamma Composition Language (introduced in Chapter 4), whereas tests for such created composite models can be defined using the Gamma Test Language (presented in Section 4.8). 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 3.2). Validation is supported by well-formedness constraints to give feedback to designers on the quality of their models at design-time (Section 3.3). Furthermore, implementation is facilitated by automatic source code generation from composition models (Section 3.4). Verification is supported by the integration of formal modeling languages, thus enabling model checking facilities. The framework also supports the back-annotation of model checking results and the generation of test cases (Section 3.5).

3.2 Integrating Engineering Models

The integration of engineering models is realized via model transformations, which map high-level behavioral models to the statechart language of Gamma. For each supported engineering language a separate model transformation needs to be defined. This is encumbered by the potentially different semantic details of such modeling languages, however, such transformations enable designers to use the functionalities of the Gamma framework without the manual mapping of their models. Furthermore, this approach can support the interaction of various engineering tools with the Gamma language as a base point, e.g., different statechart implementation can be executed side by side in the same Gamma composite system. Moreover, formal verification on the mapped models can be executed without the definition of an additional transformation to a low level analysis language.

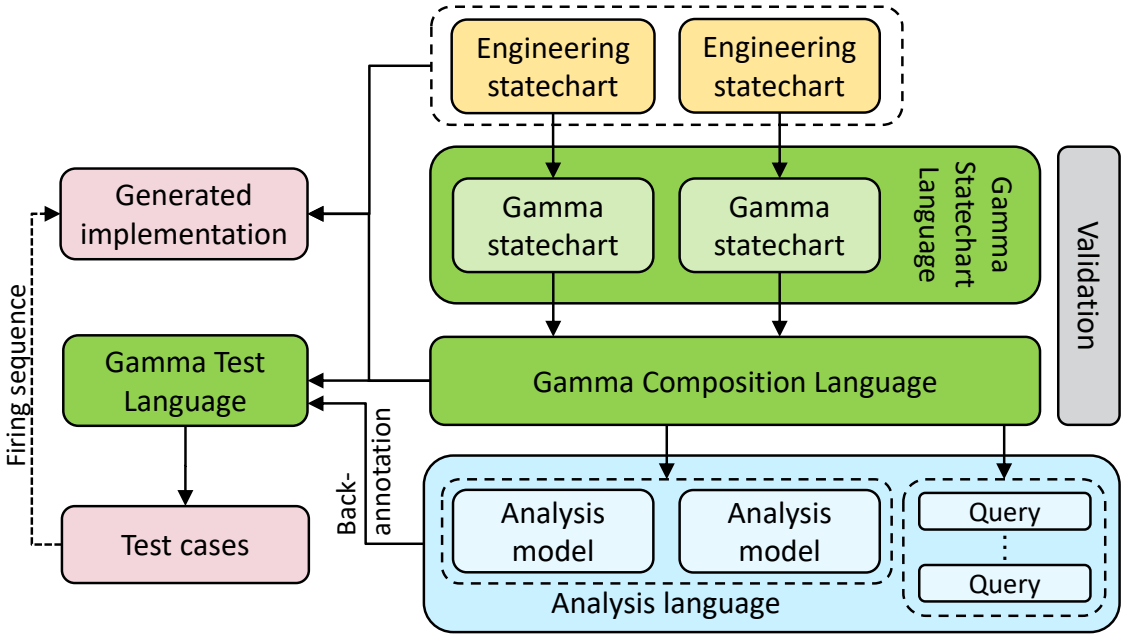

Figure 3.1: The functionalities of the Gamma framework.

During the model transformations, it is very important to create and maintain *model trace* files that store the associations between the elements of the source model and the target model. These model traces enable further functionalities of the Gamma framework regarding models and their transformations.

Currently, YAKINDU Statechart Tools (SCT) is integrated into the framework as an engineering tool. All rules of the Yakindu–Gamma transformation can be found in [9]. Moreover, the integration of another engineering tool, called MagicDraw, is in process, although it is not the work of the author.

3.3 Validation

Validation is a static analysis technique that gives feedback on the quality of created models in design time. Validation 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 approximately thirty formally defined graph patterns that specify ill-formedness constraints. Such constraints include unused variables, race-conditions and enshadowing transitions (transitions of a child and a parent state with the same trigger). If a statechart model under validation violates any constraints, the designer is notified design time with the display of the incorrect element. The validation rules can be found in [9].

The composition of statechart components is also an error-prone process which can be aided with static analysis techniques. The ill-formedness constraints are specified as graph patterns on this level as well. Validation rules on this level realize the well-formedness constraints presented in Chapter 4, that is, the validation of model imports, port bindings, channel constructions and execution order of components in cascade models.

3.4 Code Generation

The Gamma framework supports automatic source code generation from Gamma models. Currently, the Java programming language is supported, relying on the object-oriented paradigm (see Section 5.4). Interface definitions are generated from Gamma interfaces, and composite component implementations are generated from Gamma composite components defined in the composition modeling phase. The interfaces are realized by the corresponding port objects of the generated component implementations. As a result, users can interact with the generated component objects through the well-defined, familiar interfaces.

Composite component implementations wrap the necessary statechart implementations and subordinated composite components and construct the required connections (channels) between them. Thus, wrapped components are able to communicate with each other by dispatching and receiving event objects. It is essential that the behavior of composite components conforms to a rigorous semantics of the Gamma Composition Language introduced in Section 4.7. Furthermore, composite component implementations support extensibility by 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.

It is important to note that the generated composite component implementations are independent from the necessary statechart implementations and communicate with them only via a well-defined interface. Thus, the used statechart implementation is exchangeable, various statechart implementations can be introduced to the framework without the rework of the code generator. Currently, Yakindu is integrated into the framework and provides the statechart implementation. Nevertheless, the development of a Gamma statechart code generator with the support of additional programming languages is in process.

3.5 Verification

The Gamma framework provides multiple verification functionalities: 1) *formal verification* by integrating model checkers and 2) *testing* by generating test suites based on the model under verification.

3.5.1 Formal Verification

Formal verification in the Gamma framework 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 definition of model transformations that map composite system models to the formal domain. The transformation rules can be found in [9]. Similarly to the integration of engineering models, the creation of model traces during model transformations is crucial in this feature too. 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 supported by auxiliary model components responsible for the conduction of interactions among components, e.g., a scheduler automaton controlling the execution of Gamma components.

To utilize the model checking facilities, in addition to the formal model, *temporal logic expressions* need to be constructed that can be considered as the formalizations of system

requirements. As the specification of such expressions is cumbersome, their construction is supported by a graphical interface with fillable patterns (see Section 5.3.2). 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 resulting *execution trace*, describing the sequence of states leading to the erroneous state, can be interpreted in a familiar domain.

Back-annotation

Back-annotation is the process of automatically mapping the results of the formal verification of an engineering model back to the engineering domain. It can be considered as a reverse model transformation, since it derives information regarding the corresponding source model using the results of the verification carried out on the formal model. These verification results are typically represented as an *execution trace* (not to be confused with the model traces mentioned earlier), i.e., consecutive steps leading to a certain system state, by model checkers after discovering a proof or counterexample proving/violating a certain system requirement [41].

Gamma aims to completely hide the underlying formal modeling formalisms from users, and thus, provides automated support for the back-annotation of UPPAAL traces to the execution traces of the Gamma models. This way, the execution traces can be analyzed in a familiar domain, which facilitates the easy correction of flaws. For this purpose, Gamma is able to parse textual traces generated by UPPAAL. From these traces the subsequent states and transition firings of the UPPAAL model can be obtained. The previously mentioned model traces generated during the model transformations are crucial here, as they provide the necessary information for the mapping of states and transitions of the UPPAAL model to the Gamma model. The result of the back-annotation is a textual model (human readable textual file describing the execution trace) conforming to the semantics of the Gamma Test Language (Section 4.8). It describes the events the Gamma model receives from the environment, the state of the Gamma model and the events it produces for the environment in each turn during the execution. This execution trace model can be

- utilized by the user to analyze the behavior of their model, e.g., in case of a violated system requirement and,
- as it is a valid test file, it can be passed to the test generator module of Gamma to produce a test for validating the design workflow of the Gamma framework, or a manual implementation of the system.

3.5.2 Test Generation

Gamma uses the UPPAAL model checker to generate tests, which can be considered as a model-based testing technique. The model-based approach of software testing includes the creation of an abstract model, in this case a Gamma model, which is used to automatically create test cases. The idea of using model checkers in testing is to interpret proofs or counterexamples as test cases [42, 43]. The main challenge is to force the model checker to systematically create sets of such proofs/counterexamples that can be used as a complete test suite. Gamma generates tests based on the *states* and *transitions* of its models.

As mentioned in Section 3.5.1, Gamma uses the Gamma Test Language to represent execution traces of Gamma models. These execution traces are easily transformable to

JUnit classes if the implementation of the model under test is available (3.4). Events raised on the model by the environment become statements, whereas model states and events raised by the model are transformed to assertions in the JUnit test cases. To support the easy differentiation between subsequent execution steps of the model, each execution step is transformed to a single method. The generated JUnit classes test whether the component implementation 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 validated for a particular trace:

- Yakindu to Gamma transformation,
- Gamma to UPPAAL transformation,
- automatic Java code generation from composite components,
- integrated statechart implementations,
- automatic 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, thus, a best-effort means to "verify the verifier tool". The Gamma framework builds on model transformations, the correctness of which is undecidable in general, therefore, formal proof on their correctness cannot be given. However, by generating and verifying witnesses, the validity of each proof/counterexample recovered during model checking can be proven, that is, the framework is well-functioning in certain cases (no assertion error is raised during the execution of the witness) or, in case a fault is present in the framework (the execution of the witness leads to an assertion error), disproved. Note that if certain behavior is absent, e.g., a state is not reachable, the aforementioned technique cannot be utilized.

State-coverage

The Gamma framework is able to automatically generate a state-covering test-suite based on a particular Gamma model. During the model transformation to UPPAAL, temporal logic expressions are automatically generated based on the *state* model elements of the particular Gamma model. Each expression describes a state reachability criterion with the following UPPAAL query syntax:

E<> component.state.

The UPPAAL model checker can parse these expressions without modification. For each expression, UPPAAL tries to discover a path (sequence of transitions) for the specified state. If the specified state is reachable, a proof is generated that can be back-annotated to the Gamma model and transformed to a JUnit test. If the state is not reachable, then possibly a fault is discovered, as states are generally created to be assumed (with the exception of error states). Taking all state reachability criteria, a test suite with full state-coverage of the Gamma model can be generated.

Transition-coverage

In addition to state-coverage, the Gamma framework is able to generate a transition-covering test-suite based on a particular Gamma model. During the model transformation

to UPPAAL, an integer ID is assigned to each Gamma transition as well as a (single) integer variable is generated in the UPPAAL model. The *integer variable* is set to the ID of the transition if the corresponding transition fires (assignment action of the transition). A temporal logic expressions can be generated for each transition, all of which describe a variable value reachability criterion for the integer variable with the following UPPAAL query syntax:

E<> integer-variable == transition-id.

Similarly to the state-coverage test suite, UPPAAL tries to discover a path, on the end of which the given variable is set to the ID of the corresponding transition. The specified variable can be set to the value, only if the corresponding Gamma transition fires. Note that in the UPPAAL model a single transition is fired at a time, even if they are contained by orthogonal regions. a In conclusion, taking all variable value criteria, a test suite with full transition-coverage of the Gamma model can be generated.

Chapter 4

The Gamma Modeling Languages

This chapter introduces the modeling languages of Gamma, which serve as the basis of the framework. The design and formalization of these languages can be considered as the most important theoretical result of this work, and thus, the most important novelty of Gamma 2.0. The following sections present the modeling languages, that is, the constraint language, the interface language, the statechart language, the composition language (including its formal semantics) and the test language, through a railway-themed running example. These languages are closely integrated, the dependencies between them are depicted in Figure 4.1.

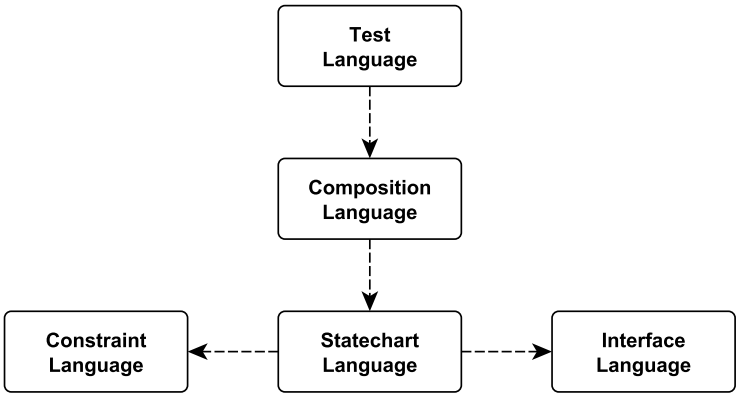

Figure 4.1: Dependencies between the modeling languages of the Gamma framework.

4.1 Running Example: Railway Safety System

Concepts of the modeling languages of the Gamma framework are demonstrated on the safety-logic model of the MoDeS^3 (Model-based Demonstrator for Smart and Safe Systems) [44] project.[1] The project imitates a railway transportation system consisting of track elements, that is, *sections* and *turnouts*, and *trains* moving on the track. The states of the turnouts (straight or divergent) and the motion of the trains (forward or backward and speed) are controlled externally by users.

A great challenge of the project was to design a distributed controller, which implements a safety-logic (introduced in [9] in detail) that can prevent the collision of trains by ensuring

[1] Project repository with link to webpage: `github.com/FTSRG/BME-MODES3`

that there is at most one train on every section at any moment. The safety-logic was designed on the basis of statecharts. Two models were created for the track elements, a section and a turnout statechart. Both statechart models detect signals of the environment, that is, the presence and absence of trains, communicate with neighboring track elements, to implement the safety-protocol, and send signals to the environment indicating dangerous situations, e.g., trains are close and they must be stopped, or the absence of danger, e.g., trains can proceed. The distributed controller is created as the composition of such section and turnout statechart model instances.

Figure 4.2 depicts the MoDeS^3 track, which consists of six turnouts and twenty five sections. The distributed controller runs on six microcontrollers, each supervising a single turnout and multiple sections. The turnout and the set of sections supervised by a particular microcontrollers is called a *zone*. The six microcontrollers have to communicate with each other, i.e., share information of their zones, to obtain a global view of the railway and prevent collisions.

4.2 Packages

Packages are the root elements of Gamma models. They support the declaration of constants (Section 4.3) in addition to the definition interfaces (Section 4.4) and Gamma components (Section 4.6.2). Furthermore, packages can depend on other packages by importing them. When a particular package is imported, its elements, e.g., statechart definitions, composite components and interfaces become visible from the importer package. Consequently, the imported components and interfaces can be used in the definitions of the importer package.

An example package is depicted in Figure 4.3. Additionally, an asynchronous adapter component *Z1Adapter* (see Section 4.6.4) is defined in the example.

Note that the imports can be defined by a path relative to the importer file, that is, the imported package is first searched for in the folder of the importer file, and if it is not found, it is searched for in the parent folder of the importer file, etc. Also, the *gcd* extension of the files does not have to be stated explicitly.

Constant *QUEUE_CAPACITY* could be referred to in any of the component definitions. If another package imported *z1_adapter*, then the importer could refer to and reuse all the elements of this particular package.

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

4.3 The Constraint Language

The Gamma Constraint Language serve as the basis of the modeling languages of Gamma, it does not depend on any other parts. It supports the definition of *constraints*, which is a general concept for type definitions, constant, variable, structure and function declarations and the specification of expressions. The functionality of the constraint language is needed by the GSL to declare and handle variables in statecharts. *Integer*, *natural*, *boolean*, *real* and *enumeration* types are supported by the language. Furthermore, there are about forty expression types in the language, including literal, logical, arithmetical and assignment expressions.

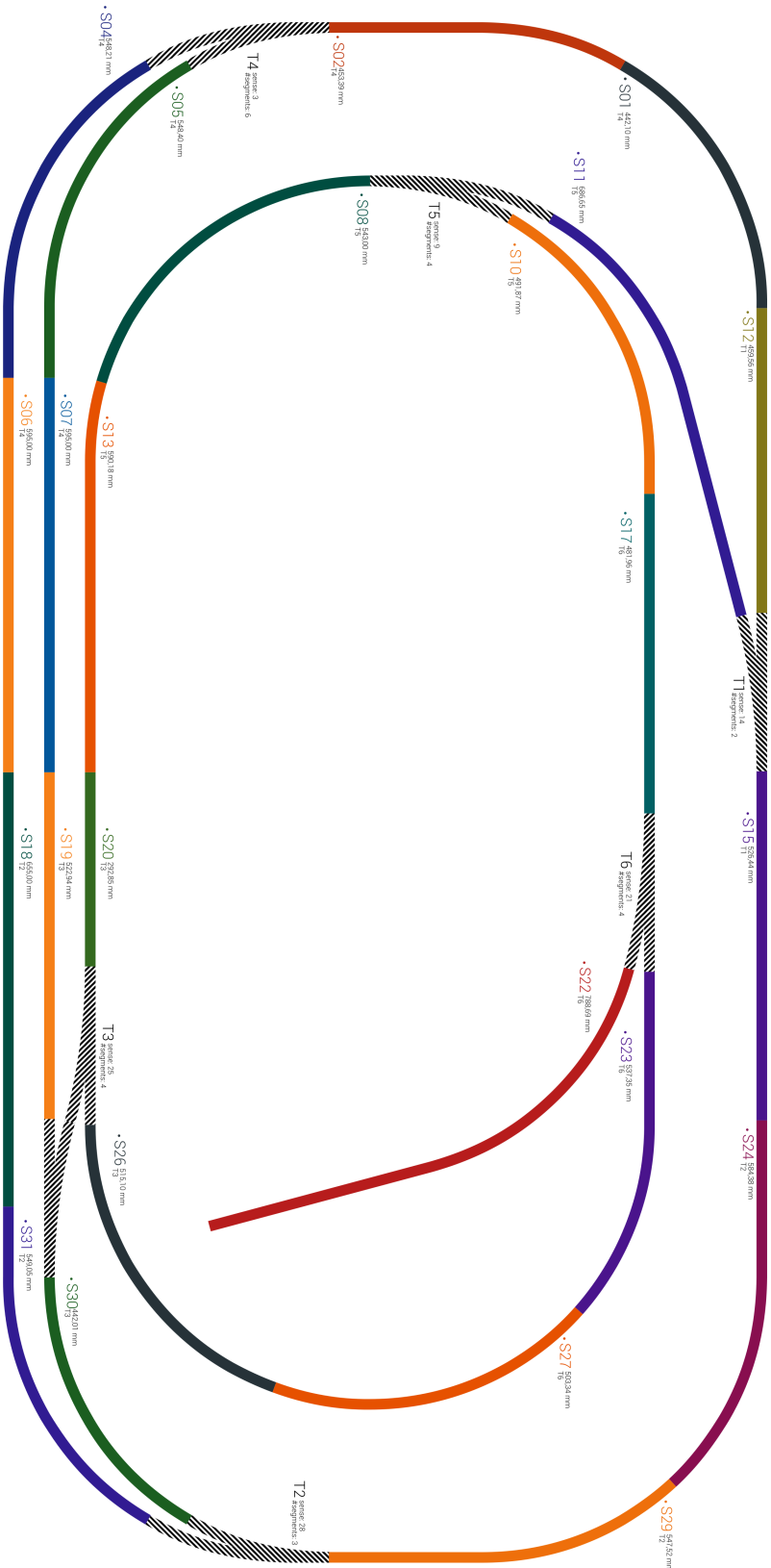

Figure 4.2: The layout of the MoDeS^3 track.

```
package z1_adapter
// Importing other packages
import "model/Components/Z1"
// Constants can be resued in the definitions
const QUEUE_CAPACITY : integer := 16
// Component definition
async Z1Adapter of Z1 [
  ...
] {...}
```

Figure 4.3: A package importing composite component *Z1* of the MoDeS^3 safety-logic model, defining a constant declaration and an asynchronous adapter component *Z1Adapter*.

The basics of this language were designed by fellow researchers of the Fault Tolerant Systems Research Group and we defined only minor additions to it. Table 4.1 summarizes the relevant operators in the current state of the Gamma framework.

Table 4.1: 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	Antivalence	xor	logical XOR	N-ary
4	Conjunction	and	logical AND	N-ary
5	Negation	not	logical NOT	Right-to-left
6	Equality	=, /=	equality/inequality	Right-to-left
7	Relational	<, >, <=, >=	comparison operators	Left-to-right
8	Addition	+, -	addition, subtraction	Left-to-right
9	Scaling	*, /	multiplication, division	Left-to-right
10	Modulo	mod	modulo	Left-to-right
11	Euclidian division	div	Euclidean division	Left-to-right
12	Sign	+, -	unary plus, unary minus	Left-to-right

4.4 The Interface Language

The Gamma Interface Language (GIL) 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. The direction of an event can be *in*, *out* or *inout*; the latter represents events that can be used as both *in* and *out* events, practically a syntactic sugar. 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 declared on its parents and is permitted to contain additional ones. This way, a particular interface might be used where its ancestor is expected. An example interface definition can be seen in Figure 4.4.

The interfaces used in the safety-logic model of MoDeS^3 are depicted in Figure 4.5. Interface *Protocol* contains events that are dispatched and received by the models of track

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

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

Figure 4.4: Example Gamma interfaces.

elements, that is, sections and turnouts, to implement the safety-logic protocol. *SectionControl* contains events denoting that a certain section needs to be disabled (no movement is allowed on the certain section) or enabled (movement is allowed). The events of *TurnoutControl* can be used to change the state of a certain turnout. Finally, the events of *Train* denote the arrival end departure of a train on a certain section.

Note that in this model we follow the convention of declaring each event as an *out* event on the interfaces. However, the actual direction of the events on communicating ports are going to be defined by the interface realizations (see Section 4.4.1).

```
interface Protocol {
  out event occupied
  out event unoccupied
  out event go
  out event stop
}

interface SectionControl {
  out event enable
  out event disable
}
```

```
interface TurnoutControl {
  out event turnoutStraight
  out event turnoutDivergent
}

interface Train {
  out event occupy
  out event unoccupy
}
```

Figure 4.5: Interfaces in the safety-logic model of MoDeS^3.

4.4.1 Endpoint Elements

In addition to interfaces and events, GIL supports the definition of *port* and *interface realization* elements, which facilitate the communication of Gamma 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 interface definition *Descendant* in Figure 4.4.

- Provided mode: ports dispatch and receive events according to the direction they have been declared on their parent interfaces. In the current example, the component owning the realizing port will be able to dispatch event *descendantEvent* (*out* or

inout events), and receive events *baseEvents* and *descendantEvent* (*in* or *inout* events) through this port. Such a port would be defined as:

```
port ProvidePort : provides Descendant
```

- Required mode: 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 *Descendant* in required mode will be able to dispatch events *baseEvents* and *descendantEvent* (declared *in* or *inout* events in the interface), and receive event *descendantEvent* (declared *out* or *inout* events) through this port. Such a port could be defined as:

```
port RequirePort : requires Descendant
```

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 *1)* the interface realization mode is provided and *2)* 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. Thus, required ports are not able to dispatch events to the broadcast port, so no congestion, and thus, the potential loss of events, will occur (see the semantics and well-formedness constraints of channels in Section 4.6.1). As demonstrated in Figure 4.5, we heavily rely on the feature of broadcast interfaces in the safety-logic model of MoDeS^3.

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, as presented in Figure 4.5, it is possible to use only out events on every interface – then provided mode is "output" mode and required mode is "input" mode.

4.5 The Statechart Language

This section introduces the Gamma Statechart Language (GSL), which supports the definition of statecharts. Generally, this language provides a common basis for all functionalities of the Gamma framework. It is given a denotational semantics described in [9] by mapping Gamma model constructions to the elements of a formal timed automaton implementation. This section presents the most important elements of the language via the simplified version of the new *Section* model of the MoDeS^3 safety-logic (see Figure 4.6).

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 introduced in Section 4.3, where the Gamma Constraint Language is presented. Furthermore, the variables can be given an initial value using an *expression*.

```
package section
import "model/Interface/Interface"
statechart SectionStatechart [
  // Ports, the same was as in the GCL
  port ProtocolInCW : requires Protocol,
  port ProtocolOutCW : provides Protocol,
  port ProtocolInCCW : requires Protocol,
  port ProtocolOutCCW : provides Protocol,
  port SectionControl : provides SectionControl,
  port Train : requires Train
] {
  // Boolean variables
  var isDisabled : boolean
  var isOccupiedCW : boolean
  var isOccupied : boolean
  var isOccupiedCCW : boolean
  // Main region
  region main_region {
    // Initial state, pseudo state
    initial Entry
    // States
    state Stable
    state WaitForCW
    state WaitForCCW
    // Choices, pseudo states
    choice Choice1
    choice Choice2
    ...
  }
  // Transitions, with (complex) triggers, guards and actions
  transition from GlobalState to Choice1 when ProtocolInCCW.unoccupied /
    isOccupiedCCW := false
  transition from Choice1 to GlobalState [not isOccupied and isOccupiedCW] /
    raise ProtocolOutCW.go
  transition from Choice1 to GlobalState [isDisabled] /
    raise SectionControl.disableSection
  transition from Choice1 to GlobalState [else]
  // Logical releation of triggers
  transition from GlobalState to Choice2 when ProtocolInCW.go &&
    !(ProtocolInCCW.stop || ProtocolInCW.stop)
  transition from Choice2 to WaitForCCW [isDisabled] /
    raise ProtocolOutCCW.occupied
  transition from Choice2 to GlobalState [else] /
    raise SectionControl.enableSection; isDisabled := false
  ...
}
```

Figure 4.6: The simplified *Section* model in the MoDeS^3 safety-logic model presenting the important elements of the GSL.

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 transitions.

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.

Entry state An entry state can be either an **initial state** or **history 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. A history state can be either *shallow* or *deep*. They are used to remember the last active state of their parent regions. In case of shallow history states, 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. *Composite* states extend *simple* states with the ability of containing one ore more regions. If a particular state contains multiple regions, they are called orthogonal.

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 orthogonal. 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 corresponding trigger is present, *3)* its guard (if it has one) evaluates to true and *4)* no transition is enabled on a higher hierarchy level[2]. 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 used for splitting transitions. Each time a choice state is entered, all guards of its outgoing transitions are evaluated in a non-deterministic order. If a guard evaluates to true, the corresponding transition fires and its actions are executed. An outgoing transition of a choice state can be targeted to another choice state. If such a transition fires, its actions are executed before evaluating the guards of the outgoing transitions of the target choice state. Therefore, choice states can be used to create dynamic conditional branches as well as to avoid "code" duplication (trigger and action specifications). It is important to note, that GSL, for the sake of formal verification, does

[2]Note that this semantics is different from the semantics of UML statecharts, as in UML statecharts transitions on the lowest hierarchy level have the highest priority.

not support the definition of directed cycles based on choice states, that is, the series of choices connected by transitions must have a topological ordering.

Triggers Triggers specify events on which certain executions can be initiated, e.g., a transition can be fired. The GSL supports the definition of *simple* or *complex* 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:

- *Port event reference* is an event indicating the reception of a particular event through a particular port.
- *Any port event reference* is an event indicating the reception of any kind of event through a particular port.
- *Timeout event reference* is an event indicating that a certain timeout has expired.
- *Clock tick reference* is an event indicating that a particular clock has emitted a tick. In the current version of the framework, clock declarations and clock tick references can be used only in asynchronous adapters (see Section 4.6.4).

Complex triggers consist of simple triggers and describe the relation of multiple triggers as logical relations. The presence or absence of signals denote the logical value of simple triggers, that is, if a certain signal is present, the corresponding trigger evaluates to true, otherwise false. The logical relations with ascending precedence are the following: *Equal*, *Imply*, *Or*, *Xor* and *And*. Also, *Negation* is supported, that is, execution can be initiated on the absence of a certain event. Complex triggers may initiate a particular execution only if the corresponding logical relation evaluates to true.

4.6 The Composition Language

The Gamma Composition Language (GCL) supports the definition of communicating composite systems built from individual components. By building on the GIL (Section 4.4) and GSL (Section 4.5), GCL supports the definition of interfaces and associated ports, which enable individual components to act as communicating endpoints. The communication between endpoints is supported by channels, responsible for connecting port instances (Section 4.6.1). Finally, the hierarchical composition of components is supported by various component types (Section 4.6.2). The following sections introduce the role of the elements of the GCL in addition to their textual syntax and validation rules.

4.6.1 Communication Elements

This section introduces the *instance port reference*, *port binding* and *channel* elements of the GCL.

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 (see Section 4.6.2). An example instance port reference is depicted in Figure 4.7.

```
S12 . ProtocolOutCCW
```

Figure 4.7: Referring to port *ProtocolOutCCW* of section model instance *S12* in the MoDeS^3 safety-logic model.

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 mapping of the system ports of a composite component, which can be regarded as "exterior" or "public" ports of the composite component, and the ports of constituent component instances. Therefore, it refers to a single system 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. An example port binding is depicted in Figure 4.8.

```
bind S12ProtocolOutCCW -> S12 . ProtocolOutCCW
```

Figure 4.8: Binding system port *S12ProtocolOutCW* of composite component *Z1* (defined in Figure 4.12) to port *ProtocolOutCW* of contained section model instance *S12* in the MoDeS^3 safety-logic model.

Well-formedness constraints: *1)* a system port of a composite component must be mapped to a single port of a constituent component instance; *2)* a port of a constituent component instance can be connected to at most one system port; *3)* the system port and the port of the component instance must realize the same interface in the same realization mode.

Channel 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: *simple* channels and *broadcast* channels.

- Simple channels support the connection of a single port providing and a single port requiring the same interface. As explained in Section 4.4.1, this design is valid and safe as they handle the same events with the appropriate directions. A simple channel example is depicted in Figure 4.9.

```
channel [ S12 . ProtocolOutCW ] -o)- [ T1 . ProtocolInStraight ]
```

Figure 4.9: A simple channel connecting provided port *ProtocolOutCW* and required port *ProtocolInStraight* of contained section model instance *S12* and turnout model instance *T1*, respectively, in composite component *Z1* (defined in Figure 4.12) of the MoDeS^3 safety-logic model.

Well-formedness constraint: in case of synchronous and cascade composite components (see 4.6.3) a non-broadcast port must not be referred to in more than one

channel or port binding (to avoid congestion and potential loss of events, since synchronous components do 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 example is depicted in Figure 4.10.

```
channel [ train.Train1 ] -o)-
  [ lowerLevelModel.Train1, higherLevelModel.Train1 ]
```

Figure 4.10: A broadcast channel connecting broadcast port *Train1* of component instance *train* to the required ports *Train1* of component instance *lowerLevelModel* and *higherLevelModel* in the verification model of the MoDeS^3 safety-logic (see Figure 6.12).

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 (but provided ports may).

One might ask the question whether two ports can be connected if they do not realize the same interface, but the realized interfaces contain the same number of events with the corresponding parameters and directions. The answer is *no*, as both interfaces and events are strongly typed, that is, the identity of an event is also defined by the interface it is declared on. This way, the compatibility of ports can be checked solely on the basis of interface types.

4.6.2 Components

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. 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 communication elements introduced in Section 4.6.1.

Component instance 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 shall 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. Furthermore, 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. An example component instance *S12* is depicted in Figure 4.11. Note that whether the component is synchronous or asynchronous is implicitly defined by the type.

```
component S12 : Section
```

Figure 4.11: A component instance definition *S12* of type *Section* in the MoDeS^3 safety-logic model.

Well-formedness constraint: each parameter of the component type must be bound to a value with an appropriate type.

4.6.3 Synchronous Components

This section introduces the synchronous components of the GCL, including their informal behavioral semantics. Synchronous components can be either composite components (*synchronous composite component* and *cascade composite component*) or atomic components (*statechart definition* introduced in Section 4.5).

Synchronous component Synchronous components represent systems that communicate in a synchronous manner using *signals*. They are executed in a lockstep fashion. Their execution is scheduled by a scheduler, which can be a wrapping asynchronous component or even a custom scheduler implementation. The execution of a synchronous composite component conforms to a turn-based semantic. A turn is called a *cycle*. In a cycle synchronous components process incoming signals and produce output signals in accordance with their internal states. Output signals are present for a single execution period only, i.e., another execution might produce different output signals overwriting the output signals of the previous one. The semantics of synchronous components was designed in accordance with the synchronous-reactive composition semantics presented in Section 2.4.

Synchronous composite component The execution of a synchronous composite component comprises the execution of its contained component instances. When a single component instance is executed it may *1)* process signals received in the last execution cycle, *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 *in the next cycle* (others or itself).

In each cycle all component instances of the particular composite component are executed. As (being a key feature of synchronous composite components) contained components *cannot affect* each other in a single execution cycle, the execution order of contained components does not matter. The results of an execution cycle is the same regardless of the execution order of the components.

In Figure 4.12 synchronous composite component *Z1* describing Zone No. 1 in the MoDeS^3 safety-logic model is depicted in a single listing. Note how the previously introduced end-

point and communication elements can be used to establish a strongly-coupled composite component.

Cascade composite component Cascade composite components are structurally similar to 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 in a specific order, which can be based on either an *execution list* defining the order of the execution of components (has to be defined explicitly), or the declaration order of the component instances (default). The execution list can contain a particular component instance one or more times, that is, a particular component instance can be executed multiple times in a single cycle. If no execution list is defined, each component instance is executed a single time, in their declaration order.

When a component is executed, it processes all incoming signals and produces signals in accordance with its internal state. However, the effect of a signal is observable *immediately* in the same execution cycle by other component instances, and not in the next one as in synchronous composite components. Accordingly, signals sent through feedback connections, i.e., when a component instance sends a signal to another one that comes earlier in the execution order, are observable in the *next* execution cycle. Note that cascade and synchronous composite components are semantically incompatible, that is, there are models in both formalisms that cannot be simulated by a model in the other due to the differences in the signal transfer.

Well-formedness constraints: *1)* Two ports of the same cascade composite component instance cannot be connected, *2)* if an execution list is defined, it must contain each defined component instance at least once.

Figure 4.13 depicts a variant of synchronous composite component *Z1* by defining an execution list. Note that the definition of cascade and synchronous composite components are the same, apart from the *cascade* keyword at the beginning of the component definition and the optional definition of an execution list.

4.6.4 Asynchronous Components

This section introduces the asynchronous components of the GCL, including their informal behavioral semantics. Asynchronous components can be either atomic components (*asynchronous adapter*) or composite components (*asynchronous composite component*).

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*.

Asynchronous adapter An asynchronous adapter wraps a single synchronous component, turning it into an asynchronous component. Asynchronous adapters implicitly have all ports of the wrapped synchronous component – additional defined ports may be used for the reception of control messages.

Furthermore, an asynchronous adapter has one or more **message queue**s, which store the incoming messages of the component. Message queues have multiple attributes:

```
package z1
import "model/Interface/Interface"
import "model/Section/Section"
import "model/Turnout/Turnout"
sync Z1 [
  // Ports on the borders of the zone
  port T1ProtocolInDivergent : requires Protocol,
  port T1ProtocolOutDivergent : provides Protocol,
  port S15ProtocolInCW : requires Protocol,
  port S15ProtocolOutCW : provides Protocol,
  port S12ProtocolInCCW : requires Protocol,
  port S12ProtocolOutCCW : provides Protocol,
  // Control ports, needed for each contained component
  port S15Control : provides SectionControl,
  port T1Turnout : requires TurnoutControl,
  port S12Control : provides SectionControl,
  // Train ports, needed for each contained component
  port S15Train : requires Train,
  port T1Train : requires Train,
  port S12Train : requires Train
] {
  // Section and turnout components
  component S15 : SectionStatechart
  component T1 : TurnoutStatechart
  component S12 : SectionStatechart
  // Binding system ports on the border to the elements
  bind T1ProtocolInDivergent -> T1.ProtocolInDivergent
  bind T1ProtocolOutDivergent -> T1.ProtocolOutDivergent
  bind S15ProtocolInCW -> S15.ProtocolInCW
  bind S15ProtocolOutCW -> S15.ProtocolOutCW
  bind S12ProtocolInCCW -> S12.ProtocolInCCW
  bind S12ProtocolOutCCW -> S12.ProtocolOutCCW
  // Binding control ports
  bind S15Control -> S15.SectionControl
  bind T1Turnout -> T1.TurnoutControl
  bind S12Control -> S12.SectionControl
  // Binding train ports
  bind S15Train -> S15.Train
  bind T1Train -> T1.Train
  bind S12Train -> S12.Train
  // Connecting elements of the zone
  channel [ S15.ProtocolOutCCW ] -o)- [ T1.ProtocolInTop ]
  channel [ T1.ProtocolOutTop ] -o)-[ S15.ProtocolInCCW ]
  channel [ S12.ProtocolOutCW ] -o)- [ T1.ProtocolInStraight ]
  channel [ T1.ProtocolOutStraight ] -o)- [ S12.ProtocolInCW ]
}
```

Figure 4.12: The definition of Zone No. 1 in the MoDeS^3 safety-logic model.

```
package z1
  ...
cascade Z1 [
  // Ports on the borders of the zone
  port T1ProtocolInDivergent : requires Protocol,
  port T1ProtocolOutDivergent : provides Protocol,
  ...
  ] {
  // Section and turnout components
  component S15 : SectionStatechart
  component T1 : TurnoutStatechart
  component S12 : SectionStatechart
  // Execution list defining the execution order of component instances
  execute T1, S15, S12, T1, S15, S12
  // Binding system ports
  ...
  // Connecting elements of the zone
  ...
}
```

Figure 4.13: The cascade composite component variation of Zone No. 1 in the MoDeS^3 safety-logic model, in which each component instance is executed twice in an execution cycle.

- *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. Note that the absence of this attribute would imply a queue with a potentially unlimited size, thus, a model with an infinite state space. Therefore, the capacity attribute is essential in the 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, where higher values represent higher priorities.
- *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, thus hierarchical filters are enabled.

During execution, messages are retrieved from messages queues *one by one*. 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 adapter component, the message does not get transmitted.

An asynchronous adapter 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 adapter 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 reach a fix point and no additional steps can be taken. Potentially, such an execution cycle can consist of an infinite number of steps, therefore, the verification of such models is not supported. Nevertheless, this feature is useful in the implementation of asynchronous adapters.
- *Reset*: the synchronous component returns 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 synchronous component.

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.

To demonstrate the flexibility of this control specification-based approach, we present two different execution semantics, with the reuse of the *Z1* synchronous composite component model.

- In Figure 4.14 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 [45].
- In Figure 4.15 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.

Asynchronous composite component 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 to asynchronous component instances that must refer to an asynchronous component as type. It is important to note that contained composite instances cannot have synchronous components as types. In such cases, asynchronous adapters can be used, which assign asynchronous behavior to synchronous components.

In Figure 4.16 the entire MoDeS^3 safety-logic is defined as an asynchronous composite component, which contains the wrapped synchronous zone models.

4.6.5 Summary

As a summary, Table 4.2 describes the component types that are supported by the GCL in terms of *synchronousness* and *compositeness*.

Figure 4.17 presents the containment hierarchy of an example composite model. Note that the synchronous and asynchronous domain can be bridged only by asynchronous adapters. Furthermore, the leaves, which are the basic building blocks of the behavior of composite

```
package z1_adapter
import "model/Z1/Z1"
const QUEUE_CAPACITY : integer := 16
async Z1Adapter of Z1 [
  // No additional ports on the adapter
] {
  // Run the wrapped component on any kind of received event
  when any / run
  // Storing protocol messages in a higher priority queue
  queue protocolMessages (priority = 1, capacity = QUEUE_CAPACITY) {
    T1ProtocolInDivergent.any, T1ProtocolOutDivergent.any,
    S15ProtocolInCW.any, S15ProtocolOutCW.any,
    S12ProtocolInCCW.any, S12ProtocolOutCCW.any
  }
  // Storing other control messages in a lower priority queue
  queue controlMessages (priority = 2, capacity = QUEUE_CAPACITY) {
    S15Restart.any, S15Control.any, T1Turnout.any, S12Restart.any,
    S12Control.any, S15Train.any, T1Train.any, S12Train.any
  }
}
```

Figure 4.14: The UML-like asynchronous adaptation of Zone No. 1 in the MoDeS^3 safety-logic model.

```
package z1_adapter
import "model/Z1/Z1"
async Z1Adapter of Z1 [
  // No additional ports on the adapter
] {
  // Clock emitting a tick every millisecond
  clock millisecondClock(rate = 1 ms)
  // Run the wrapped component on clock events
  when millisecondClock / run
  // Storing clock messages in a higher priority queue
  queue clockMessages (priority = 0, capacity = 8) {
    millisecondClock
  }
  // Storing other messages in lower priority queues
  ...
}
```

Figure 4.15: The timed asynchronous adaptation of Zone No. 1 in the MoDeS^3 safety-logic model.

Table 4.2: Component types supported by the Gamma composition language.

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

```
package modes_track
import "model/Interface/Interface"
import "model/Z1/Z1Adapter"
...
import "model/Z6/Z6Adapter"
async ModesTrack [
  // Turnout control ports
  port T1Turnout : requires TurnoutControl,
  ...
  port T6Turnout : requires TurnoutControl,
  // Train ports
  port T1Train : requires Train,
  ...
  port S31Train : requires Train,
  // Section control ports
  port S01Control : provides SectionControl,
  ...
  port S31Control: provides SectionControl
] {
  // The instances of the wrapped zone models
  component Z1 : Z1Adapter
  component Z2 : Z2Adapter
  component Z3 : Z3Adapter
  component Z4 : Z4Adapter
  component Z5 : Z5Adapter
  component Z6 : Z6Adapter
  // Binding system ports
  bind T1Turnout -> Z1.T1Turnout
  ...
  bind T6Turnout -> Z6.T6Turnout
  bind T1Train -> Z1.T1Train
  ...
  bind S31Train -> Z2.S31Train
  bind S01Control -> Z4.S01Control
  ...
  bind S31Control -> Z2.S31Control
  // Connecting zones
  // Z1
  channel [ Z1.T1ProtocolOutDivergent ] -o)- [ Z5.S11ProtocolInCW ]
  channel [ Z5.S11ProtocolOutCW ] -o)- [ Z1.T1ProtocolInDivergent ]
  channel [ Z1.S12ProtocolOutCCW ] -o)- [ Z4.S01ProtocolInCW ]
  channel [ Z4.S01ProtocolOutCW ] -o)- [ Z1.S12ProtocolInCCW ]
  channel [ Z1.S15ProtocolOutCW ] -o)- [ Z2.S24ProtocolInCCW ]
  channel [ Z2.S24ProtocolOutCCW ] -o)- [ Z1.S15ProtocolInCW ]
  // Z2
  channel [ Z2.T2ProtocolOutDivergent ] -o)- [ Z3.S30ProtocolInCCW ]
  channel [ Z3.S30ProtocolOutCCW ] -o)- [ Z2.T2ProtocolInDivergent ]
  channel [ Z2.S18ProtocolOutCW ] -o)- [ Z4.S06ProtocolInCCW ]
  channel [ Z4.S06ProtocolOutCCW ] -o)- [ Z2.S18ProtocolInCW ]
  // Z3
  channel [ Z3.S19ProtocolOutCW ] -o)- [ Z4.S07ProtocolInCCW ]
  channel [ Z4.S07ProtocolOutCCW ] -o)- [ Z3.S19ProtocolInCW ]
  channel [ Z3.S20ProtocolOutCW ] -o)- [ Z5.S13ProtocolInCCW ]
  channel [ Z5.S13ProtocolOutCCW ] -o)- [ Z3.S20ProtocolInCW ]
  channel [ Z3.S26ProtocolOutCCW ] -o)- [ Z6.S27ProtocolInCW ]
  channel [ Z6.S27ProtocolOutCW ] -o)- [ Z3.S26ProtocolInCCW ]
  // Z5
  channel [ Z5.S10ProtocolOutCW ] -o)- [ Z6.S17ProtocolInCCW ]
  channel [ Z6.S17ProtocolOutCCW ] -o)- [ Z5.S10ProtocolInCW ]
}
```

Figure 4.16: The high-level MoDeS^3 safety-logic model.

components, are always statechart definitions. Currently, statecharts cannot be directly wrapped by asynchronous adapters, they must be contained by a synchronous or a cascade composite component. Note that this is only a syntactical restriction and does not limit the possible behaviors of the created models.

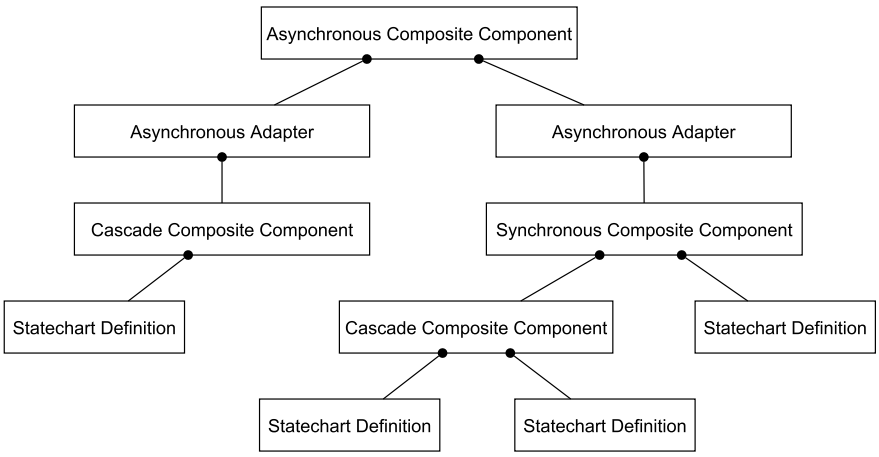

Figure 4.17: A composite model hierarchy in Gamma.

4.7 Formal Semantics of the Composition Language

This section presents the formal structures and semantics of the GCL. Subsections include short discussions about additional practical and theoretical aspects, design decisions and consequences.

The section starts with the definition of events and related structures (Section 4.7.1), then the syntactic definition of a synchronous component (Section 4.7.3) and event vectors related to signals (Section 4.7.2) are introduced. Next, synchronous composite components, as well as cascade composite components are formalized both syntactically and semantically (Section 4.7.4 and Section 4.7.5). After defining event sequences (Section 4.7.6) and asynchronous components (Section 4.7.7), asynchronous adapters and their semantics are presented (Section 4.7.8). The section concludes with the definition of asynchronous composite components (Section 4.7.9) and their semantics in terms of messages, occurrences and execution traces (Section 4.7.11). To help the reader find their way through the following pages, the Appendix lists the symbols used in the definitions along with a short description.

4.7.1 Events

Definitions of Section 4.7 considers individual events only, since ports and interfaces are syntactic sugar that facilitate the structuring of syntactic contracts. The event definition below models a specific event of a specific port on a specific component.

Definition 2. An *event* is an observable phenomenon that can occur, such as the reception of a message or the change of situation (state). Given a set of events E, the finite domain of events is defined by the domain function $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$. The domain of an event $e \in E$ is $\mathcal{D}(e)$. We say that an event $e \in E$ is *parameterized* if $|\mathcal{D}(e)| > 1$. An *instance* of an event is (e, p), i.e., the event with a specific parameter value $p \in \mathcal{D}(e)$. The

set of all event instances for a given event e is denoted by $inst(e) = \{(e, p) \mid p \in \mathcal{D}(e)\}$. In case the absence of an event is of interest, $inst_\perp(e)$ is defined as $inst(e) \cup \{(e, \perp)\}$, where $(e, \perp)$ is the "null" instance that denotes the absence of the event. Finally, the set of event instances for events in a set E is $inst(E) = \bigsqcup_{e \in E} inst(e)$ (and $inst_\perp(E)$ similarly).

Discussion An event represents a declaration only. It does not need memory in and of itself, but an instance is referred to in event vectors and messages (Sections 4.7.2 and 4.7.11), which do need memory during runtime or verification.

4.7.2 Event Vectors

In the synchronous domain, components communicate via signals. The formal structure describing signals is the event vector. An event vector can be regarded as a set of cells that can be filled with event instances, at most one instance in every cell. Event vectors are the inputs and outputs of synchronous components.

Definition 3. Given a set of events E, an *event vector* v_E is a function that assigns a (possibly "null") event instance to every event $e \in E$ such that $v_E(e) \in inst_\perp(e)$. The set of all possible event vectors is denoted by V_E.

Discussion. As mentioned before, event vectors do need memory to represent them at runtime. Events vectors can be regarded as "nullable" variables dedicated to each event holding the occurrence and the parameters of that particular event (if any).

4.7.3 Synchronous Component

The following definition specifies the formal syntactic contract of synchronous components. A synchronous component should have a set of states, a well-defined initial state, a set of input and output events (collected from ports of the component) along with their parameter domains, i.e., data type, and a deterministic transition function[3] that describes the behavior of the component, which can be specified arbitrarily.

Definition 4. A synchronous component is a tuple $\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- S is the finite 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 $I \cap O = \emptyset$. The set of all events is denoted by $E = I \sqcup O$.
- $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain function of the events.
- $T \colon S \times V_I \to S \times V_O$ is the transition function, which determines the next state and the output event vector of the component when executing it in a given state with a given input event vector. Note that this definition requires the component to have a deterministic behavior.

[3]This definition is extensible to nondeterministic components as well. However, this extension would be relevant only in case of environmental models, which is not in the primary scope of the current work.

Discussion. Synchronous components take an event vector as an input and generate an event vector as an output. Considering that the synchronous component is a statechart, the definition is closest to the Virtual Finite State Machine formalism introduced in [46]. We chose this formalism because it harmonizes with the synchronous-reactive domain – as components are executed in a lock-step fashion, there may be a need to react to multiple events or a combination of events at the same time, which can be hanndled by complex triggers. Nevertheless, the more widespread Event-Driven Finite State Machine, which is the basis of most commonly used statechart formalisms, is also suitable to describe a component by triggering to event vectors with a single "non-null" event instance only and by ensuring that they are run *every time* a signal arrives (see the asynchronous adapter in Section 4.7.8).

4.7.4 Synchronous Composite Component

Recall that a synchronous composite component is defined by instantiating a set of constituent components, exporting input and output ports (events in the formal case) by port bindings and defining channels (connecting events instead of ports in the formal case).

Definition 5. A synchronous composite component is a tuple $Ⓢ = (\mathbf{C}, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{⊖_1, \ldots, ⊖_K\}$ is the set of synchronous components constituting the composite component, each component being $⊖_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigsqcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigsqcup_{k=1}^{K} O_k$.
- $\rightleftharpoons: \hat{I} \setminus I \to \hat{O}$ 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 an exported input. We demand that for each $e \in \hat{I}$, $\mathcal{D}(e) = \mathcal{D}(\rightleftharpoons(e))$.

Discussion. Note that *port binding* elements are not present in the definition, instead *exported* events are defined. This implies that events bound together in a composite model are handled as if they were the same, they are not differentiated in any way, as they represent the same occurrence.

Semantics. To understand the semantics of synchronous composite components, i.e., its behavior as a synchronous component, recall that output signals produced by a component are sampled by other components in the next execution cycle only. To describe this behavior, we extend the combined state space of the constituent components with the last output event vector *of all constituent components*. An execution cycle is described by the emergent transition relation of the composite component.

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

- $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ is the set of potential states, derived as all possible combinations of the potential states of the constituent synchronous components and the last output event vector of every component.

- $s^0 = (s_1^0, \ldots, s_K^0, \bot_{\hat{O}})$ is the initial state, where every constituent synchronous component is in its initial state and the last output event vector $\bot_{\hat{O}} \in V_{\hat{O}}$ assigns $\bot$ to every output event ($\forall e \in \hat{O} \ : \ \bot_{\hat{O}}(e) = \bot$).
- I is the set of exported input events and O is the set of exported output events as defined in Definition 5 (remember that we denote $I \sqcup O$ by E).
- $\mathcal{D}$ 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 $e \in E$.
- 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 $v_{\hat{I}}(e) = v_{\hat{O}}(\rightleftharpoons(e))$ 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 vector of last output events.
  - The output of the composite component for each exported output $e \in O$ is defined by the output of the constituent components: $v_O(e) = v'_{\hat{O}}(e)$.

Discussion. When executing a synchronous composite component, its constituent components either react to an external input (in case of exported inputs) or to the output of a constituent component (including themselves) *from the previous execution cycle.* This prevents any interaction between the components during a single execution cycle, allowing to execute the components in an *arbitrary order*, essentially performing *partial order reduction* statically. This key feature greatly reduces the size of the state space, making the synchronous-reactive domain suitable for formal verification. Additionally, the definition enables to connect a single output to multiple inputs of components, however, an input can be connected only to a single output.

4.7.5 Cascade Composite Component

The syntactic definition of cascade composite components is the same as that of synchronous composite components, apart from the additional definition of the execution order of constituent components.

Definition 7. A cascade composite component is a tuple $\copyright = (\mathbf{C}, X, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{\ominus_1, \ldots, \ominus_K\}$ is the set of synchronous components constituting the composite component, each component being $\ominus_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $X \in \mathbf{C}^*$ is a finite *ordered sequence* (with potential repetitions) of synchronous components called the *execution sequence* specifying the components to be executed in an execution cycle.
- $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigsqcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigsqcup_{k=1}^{K} O_k$.
- $\rightleftharpoons : \hat{I} \setminus I \to \hat{O}$ 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 an exported input. We demand that for each $e \in \hat{I}$, $\mathcal{D}(e) = \mathcal{D}(\rightleftharpoons(e))$.

Semantics. Cascade composite components do not delay the internal signals between constituent components, therefore the effect of an event is computed in a single run. Signals sent to components that are not executed anymore in the current execution cycle are saved for the next cycle, just like in synchronous composite components.

Definition 8. A cascade composite component © is itself a synchronous component $©\rangle\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_1 \times \ldots \times S_K \times V_{\hat{O}}$ is the set of potential states, derived as all possible combinations of the potential states of the constituent synchronous components and the last output event vector of every component.
- $s^0 = (s_1^0, \ldots, s_K^0, \bot_{\hat{O}})$ is the initial state, where every constituent synchronous component is in its initial state and the last output event vector $\bot_{\hat{O}} \in V_{\hat{O}}$ assigns $\bot$ to every output event ($\forall e \in \hat{O} \ : \ \bot_{\hat{O}}(e) = \bot$).
- I is the set of exported input events and O is the set of exported output events as defined in Definition 7 (recall that $I \sqcup O$ is denoted by E).
- $\mathcal{D}$ 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 $e \in E$.
- The transition function is $T\big((s_1, \ldots, s_K, v_{\hat{O}}), v_I\big) = \big((s'_1, \ldots, s'_K, v'_{\hat{O}}), v_O\big)$, computed iteratively for every $X[i]$ ($1 \leq i \leq n$, $n = |X|$):
  - Let $(s_1^0, \ldots, s_K^0, v_{\hat{O}}^0) = (s_1, \ldots, s_K, v_{\hat{O}})$ (the source state).
  - Assume that $X[i] = \ominus_k$. To obtain $(s_1^i, \ldots, s_K^i, v_{\hat{O}}^i)$, we apply $T_k(s_k^{i-1}, v_{I_k})$ $= (s_k^i, v_{O_k})$ to compute s_k^i and v_{O_k}, where for all $e \in I_k$, $v_{I_k}(e) = v_I(e)$ if $e \in I$, and $v_{I_k}(e) = v_{\hat{O}}^{i-1}(\rightleftharpoons(e))$ otherwise. The state of other components $\ominus_j \in \mathbf{C}$ ($j \neq k$) remains the same ($s_j^i = s_j^{i-1}$). The last output events vector is updated with v_{O_k}: for all $e \in \hat{O}$, $v_{\hat{O}}^i(e) = v_{O_i}(e)$ if $e \in O_k$, and $v_{\hat{O}}^i(e) = v_{\hat{O}}^{i-1}(e)$ otherwise.
  - Finally, $s'_k = s_k^n$ for every $\ominus_k \in \mathbf{C}$ and $v_O(e) = v_{\hat{O}}^n(e)$ for every $e \in O$.

Discussion. The raison d'etre of the cascade composite semantic variant is twofold. First, even though it requires the same amount of memory to represent as synchronous composite components (see the definition of S), the effect of an input event on output events is computed in a single step, further compressing the state space (assuming that a composite component is stimulated in hopes of observing an output). Second, it is sometimes desired to "decorate" a component with auxiliary components such as adapters or monitors (like in the verification models of the MoDeS^3 case study in Section 6.3.4) without introducing a delay in the observable effect of an event. Furthermore, it is convenient to think in terms of pipelines, which is best expressed with cascade composite components.

One drawback of using cascade composite components is that the outputs of constituent components may overwrite each other if a particular component is run multiple times (but this is still deterministic), and all outputs of all components are emitted in a single event vector. If the temporal unfolding of the different reactions is relevant, it may be more beneficial to use a synchronous composite component.

4.7.6 Event Sequences

In the asynchronous-reactive domain, event vectors are substituted by event sequences.

Definition 9. An *event sequence* $q = \langle (e_1, p_1), \ldots, (e_n, p_n) \rangle$ is a finite, possibly empty (denoted by ε) sequence of event instances. The set of all possible event sequences for a set of events E with domain function $\mathcal{D}$ is denoted by $inst(E)^*$, while $|q|$ denotes the length of the sequence. The ith event instance in the sequence is denoted by $q[i] = (e_i, p_i)$. Finally, a permutation of a set A is a sequence $\sigma(A)$ and all possible permutations of A is denoted by $S_\sigma(A)$.

4.7.7 Asynchronous Component

Asynchronous components are syntactically very similar to synchronous components. The only difference is the definition of transitions: it is now not a function but a relation, and instead of taking and producing an event vector, it takes a single event instance and produces an event sequence chosen from the potential output sequences nondeterministically.

Definition 10. An asynchronous component is a tuple $\circledminus = (S, s^0, I, O, \mathcal{D}, T)$:

- 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 $I \cap O = \emptyset$. The set of all events is denoted by $E = I \sqcup O$.
- $\mathcal{D} : E \to \{d_1, \ldots, d_n\}$ is the domain of the events.
- $T \subseteq S \times inst(I) \times S \times inst(O)^*$ is the transition relation, which determines the next state and the sequence of output events of the component ($inst(O)^*$) 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.

Discussion. Contrary to synchronous components, the definition of asynchronous components is closest to Event-Driven Finite State Machines or the variant of statecharts defined in UML. Although currently not supported by the Gamma Statechart Language, asynchronous components could be implemented directly by statecharts. In Gamma, the current means of defining an asynchronous statechart component is to define a synchronous composite component containing a statechart, and wrap it in an asynchronous adapter (these steps may become automated in a future release).

Note that allowing a non-deterministic transition relation is necessary because the order of output events may not always be specified, e.g., in case of parallel regions. In case of synchronous components, the order of events does not matter as they are collected in an event vector. The event sequence, however, will be different depending on the internal order of raising events. This phenomenon poses challenges to both verification and code generation, and hinders the reproducibility of test cases and counterexamples. Nondeterministic behavior, however, is inherent in the asynchronous-reactive domain anyway.

4.7.8 Asynchronous Adapter

Recall that an asynchronous adapter wraps a single synchronous component and integrates it for the asynchronous domain. To do this, the *trigger predicate* with a set of trigger specifications have to be defined (see Section 4.6.4). Additional ports may also be defined. Formally, the opportunity to define multiple additional ports and events on them is only a syntactic sugar, as all of them are mapped to the *control event* introduced in the definition

below. This formalization does not include the "reset" and "run to completion" options available in control specifications, which are currently not supported by formal verification.

Definition 11. An asynchronous adapter for a synchronous component is defined as a tuple $\oplus\!\!\boxplus = (\ominus, e_c, \mathit{trig})$:

- $\ominus = (S_s, s_s^0, I_s, O_s, \mathcal{D}_s, T_s)$ is the wrapped synchronous component.
- e_c is the *control event*.
- $\mathit{trig}\colon I_s \cup \{e_c\} \to \{\top, \bot\}$ is the *trigger predicate* that given an input event returns if the wrapped synchronous component must be executed or not.

Semantics. The semantics of asynchronous adapters is defined in terms of an asynchronous component. Observed from an external component, an adapter processes input events one-by-one (just like asynchronous components in general), but may not always produce an output. The role of the adapter is to "collect" messages for the wrapped synchronous component, and when a message triggers execution, that is, $\mathit{trig}(e)$ is $\top$, feed the collected messages and emit messages created from the resulting output event vector.

Definition 12. An asynchronous adapter $\oplus\!\!\boxplus$ for a synchronous component is itself an asynchronous component $\oplus\!\!\boxplus\rangle\!\ominus = (S, s^0, I, O, \mathcal{D}, T)$:

- $S = S_s \times v_I$ is the set of potential states, each state consisting of a state of the wrapped synchronous component and a buffer input event vector collecting the incoming event instances.
- $s^0 = (s_s^0, \bot_I)$, where $\bot_I$ is the empty input vector.
- $I = I_s \cup \{e_c\}$ is the set of input events including the input events of the wrapped synchronous component and the control event. From an input vector v_I we can derive $v_{I\,s}$ as $v_{I\,s}(e) = v_I(e)$ for every $e \in I_s$.
- $O = O_s$ is the set of output events defined in the wrapped synchronous component.
- $\mathcal{D} = \mathcal{D}_s \cup (e_c \to \{\top\})$ 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 transition function is defined as $T\big((s_s, v_I), (e, p)\big) = \{(s_s', v_I')\} \times \Omega$, such that:
  - If $\mathit{trig}(e) = \bot$, then the buffer input event vector is updated such that $v_I'(e) = (e, p)$ and $v_I'(e') = v_I(e')$ for every $e' \in I$ $(e \neq e')$, and $s_s' = s_s$, while $\Omega = \{\varepsilon\}$ (as the set of possible output sequences) is the empty sequence in this case.
  - If $\mathit{trig}(e) = \top$, then the buffer input event vector is updated such that $v_I''(e) = (e, p)$ and $v_I''(e') = v_I(e')$ for every $e' \in I$ $(e \neq e')$, and s_s' should be such that $T_s(s_s, v_I'') = (s_s', v_O)$, and $v_I' = \bot_I$. $\Omega = S_\sigma(\{(e, p) \mid v_O(e) = p, p \neq \bot\})$ (as the set of possible output sequences) is every possible permutation of the "non-null" elements of the output vector.

Discussion. The order of messages between two execution-triggering messages is not relevant as long as they do not overwrite each other, so the adapter may store an event vector as a buffer instead of a message queue. In practice, the memory allocated for the input vector of the wrapped component can be reused.

The definition of asynchronous adapters is very flexible. Components like an Event-Driven Finite State Machine may be implemented by a synchronous component by declaring no additional control events, but returning $\top$ from the trigger predicate for any event (using the keyword "any"). With the help of the control event, however, it is also possible to promote the "ticks" of the wrapped synchronous component to its syntactic contract, which is the preferred way of handling even a single synchronous system in Gamma. The definition also allows mixed solutions, e.g., a component may be triggered by any external control event or by any event on one of its ports.

Note that according to the definition, the sequence of output events may be any permutation of the "non-null" events in the output vector of the wrapped component. Although consistent with the definition of asynchronous components, this is rather an underspecification than real nondeterminism – most implementations would raise output events in a fixed order, e.g., when wrapping a cascade composite component.

4.7.9 Asynchronous Composite Component

The syntactic definition of an asynchronous composite component differs from synchronous composite components only in the definition of channels. Since asynchronous components operate with event sequences, it is not a problem anymore if an input event has multiple sources, so there is no restriction on channels other than parameter compatibility.

Definition 13. An asynchronous composite component is a tuple $ⓐ = (\mathbf{C}, I, O, \rightleftharpoons)$:

- $\mathbf{C} = \{⊜_1, \ldots, ⊜_K\}$ is the set of asynchronous components constituting the composite component, each component being $⊜_k = (S_k, s_k^0, I_k, O_k, \mathcal{D}_k, T_k)$.
- $I \subseteq \hat{I}$ is the set of exported input events, where $\hat{I} = \bigsqcup_{k=1}^{K} I_k$.
- $O \subseteq \hat{O}$ is the set of exported output events, where $\hat{O} = \bigsqcup_{k=1}^{K} O_k$.
- $\rightleftharpoons \subseteq \hat{O} \times \hat{I}$ is the set of *channels* that connects inputs and outputs with no restriction apart from parameter compatibility. The set of inputs connected to an output e is denoted by $\rightleftharpoons(e) = \{e' \mid (e, e') \in \rightleftharpoons\}$. We demand that for each $e \in \hat{I}$ and $e' \in \rightleftharpoons(e)$, $\mathcal{D}(e) = \mathcal{D}(e')$. Note that $\rightleftharpoons(e)$ used as a function maps from outputs to inputs, contrary to the notation used in synchronous components, where it mapped from inputs to outputs.

Discussion. In asynchronous composite components, events are transferred in messages and processed one-by-one. It is generally assumed that components have a message queue where sent but unprocessed messages are stored.

4.7.10 External Component

The environment of an asynchronous composite component is modeled with an external component.

Definition 14. Given an asynchronous composite component $ⓐ = (\mathbf{C}, I, O, \rightleftharpoons)$, an *external component* is a tuple $ⓔ = (E_I^{ext}, E_O^{ext})$:

- $E_I^{ext} = O$ is the input events of the external component that serve as the output events of the asynchronous composite component.
- $E_O^{ext} = I$ is the output events of the external component that serve as the input events of the asynchronous composite component.

Discussion. The behavior of the external component is considered nondeterministic. Nevertheless, in future work we plan to restrict its behavior with scenario-based contracts.

4.7.11 Messages and Execution Traces

The semantics of asynchronous composition can be defined in terms of messages and occurrences. A message is defined in terms of its source and target events and its parameter.

Definition 15. Given an asynchronous composite component $⊜$ with its external component ⓔ, an asynchronous *message* is a tuple $m = (e_O, p, E_I)$:

- $e_O \in \hat{O} \cup E_O^{ext}$ is the source output event of the message, possibly coming from the environment.
- $p \in \mathcal{D}(e_O)$ is the content of the message.
- $E_I \subseteq \hat{I} \cup E_I^{ext}$ is the set of target input events of the message, possibly targeting the environment.
- If $e_O \in E_O^{ext}$ then $E_I \subseteq I$ and if $E_I \subseteq E_I^{ext}$ then $e_O \in O$, that is, external messages may arrive through exported input events, while external targets may be addressed through exported output events. If $e_O \notin E_O^{ext}$ and $E_I \not\subseteq E_I^{ext}$ then $\rightleftharpoons(e_O) = E_I$, that is, if the message is sent to another component in the same asynchronous composite component, the corresponding inputs and outputs are connected with a channel.

Let $send(m)$ denote the occurrence of creating the message in response to its source output event and $recv(m, e_I,)$ the occurrence of consuming the message on input event $e_I \in E_I$, thus, raising event e_I. The source component of a message is denoted by $src(m) = ⊜_k \in \mathbf{C}$ when $e_O \in O_k$ or $src(m) = ⓔ$ if $e_O \in E_O^{ext}$.

Furthermore, let $t = (s, e_I, s', \omega) \in T_k$ be a transition of a constituent component $⊜_k$. An occurrence of transition t is a tuple $[t] = (m_I, t, M_O)$, where:

- $m_I = (e_O, p, E_I)$ is the message *triggering* the transition, while $e_I \in E_I$.
- t is the triggered transition.
- M_O is the sequence of *raised* messages such that $|M_O| = |\omega|$ and for every $1 \leq i \leq |M_O|$, $M_O[i] = (e'_O, p', E'_I)$ such that $\omega[i] = (e'_O, p')$.

Discussion. A message is a runtime object, i.e., it has "object identity". For example, in the *ModesTrack* model (presented in Section 4.6.4) event "*Z1.S15ProtocolOutCW.go*" creates a message m with the same source and "*Z4.S24ProtocolInCCW.go*" as the target, with no parameter (the domain is a singleton set). Raising it again creates another *different* message m' but with the same content. Occurrences, such as message sending, receiving and firing transitions constitute the observable behavior of an asynchronous system, e.g., sending message m is an observable happening at a specific point in time. Occurrences enable us to define an execution trace, describing the behavior of asynchronous systems.

Definition 16. Given a totally ordered sequence of transition occurrences and message sending and receiving (that is, an *execution trace*), let $\#[t]$, $\#send(m)$ and $\#recv(m, e_I)$ denote the position of the corresponding occurrence in the ordering. The execution of an asynchronous composite component must obey the following rules (defining a partial order):

1. *(causality)* $\#send(m) < \#recv(m, e_I)$ for every message $m = (e_O, p, E_I)$ appearing in the execution trace and for every $e_I \in E_I$.

2. *(causality)* $\#recv(m_I, e_I) < \#send(m_O)$ for every transition occurrence $[t] = (m_I, t, M_O)$ appearing in the trace where $m_O \in M_O$.

3. *(message order)* If $\#send(m) < \#send(m')$ such that $src(m) = src(m')$, then $recv(m, e_I) < \#recv(m', e'_I)$ for every e_I and e'_I belonging to the same component $\ominus_k$ ($e_I \in I_k$ and $e'_I \in I_k$).

4. *(message order)* For every transition occurrence $[t] = (m_I, t, M_O)$ and for each $1 \leq i, j \leq |M_O|$ if $i < j$ then $\#send(M_O[i]) < \#send(M_O[j])$.

Discussion. The first two rules enforce causality: an occurrence cannot happen before another occurrence that caused it to happen. The third rule is a constraint on the implementation of asynchronous systems of the GCL: the communication is demanded to be reliable not only in terms of losing messages (implicitly forbidden by Rule 1), but also in terms of the order of messages. The fourth rule, on the other hand, describes the natural mapping between output event sequences and the generated message sequences. These requirements are usually not impossible to meet, while the lack of these assumptions would greatly hinder verifiability.

4.8 Gamma Test Language

The Gamma Test Language (GTL) supports the definition of execution traces regarding the components of the GCL. Such execution traces formally describe the behavior of Gamma components, that is, what states a particular component assumes (state configuration and variable values) and what events it produces in response to certain inputs (input events, time elapse and the scheduling of the component). Execution traces are important during

1. model checking, when a counterexample (or proof) is revealed and it must be constructed and presented in an intuitive way. In this case, the back-annotator module of the Gamma framework produces a GTL model;

2. testing, when the user wants to test whether the implementation of the designed component (both from the Gamma framework and 3rd-party modeling tools) actually behave as expected. In this case, either the test-generator module of the Gamma framework can produce a test suite with the necessary acts and assertions, or users can define tests manually.

As presented in Section 4.6.3, synchronous Gamma components conform to a turn-based semantics, where turns are called cycles. These cycles are represented by *steps* in an execution trace. In a particular step *acts* and *assertions* can be defined. An act can be either

- an *input event raise*, which describes the raising of an event on a system port of the particular component;
- *time elapse* describing the elapse of time in milliseconds;
- *scheduling*, which describes either
  - the scheduling of the component in case of a synchronous component, or
  - the scheduling of a contained asynchronous instance in case of an asynchronous component, as asynchronous component instances run individually.

An assertion can refer to either

- an *output event raise*, which describes the raising of an event on a system port by the particular component;
- the *value of a variable declaration*, where the value can be expressed with an arbitrary expression, or
- a *state configuration*, where the supposedly active states of a synchronous component instance are defined. Note that asynchronous component instances cannot be referred to in the description of such state configurations, because they do not have separate states: their states derive from the states of the wrapped synchronous components.

Execution traces contain every essential behavioral detail of a Gamma component as these details collectively define the precise behavior of the component. However, in case of tests some details may be unimportant, e.g., the user is interested only in the raised events of the component and not in its state configurations. In such cases, the irrelevant parts of the execution traces may be removed to allow for more general testing.

Figure 4.18 presents an execution trace of component Z1 of the MoDeS^3 safety-logic.

```
import "model/Z1/Z1"
component Z1
step {
  assert {
    // Initial "collective" state of the composite component
    states {
      // Variable values
      S12.isDisabled = false,
      S12.isOccupiedCW = false,
      S12.isOccupied = false,
      S12.isOccupiedCCW = false,
      S15.isDisabled = false,
      S15.isOccupiedCW = false,
      S15.isOccupied = false,
      S15.isOccupiedCCW = false,
      // States configurations
      S12.GlobalState,
      T1.Straight,
      S15.GlobalState
    }
  }
}
step {
  act {
    // Raising in-event occupy on port S12Train
    raise S12Train.occupy
    // 100 millisecond elapse
    elapse 100
    // Scheduling the composite component
    schedule component
  }
  assert {
    // Out event raised by the composite component
    raised S12Control.enableSection
    states {
      // Variable values changed compared to the previos step
      S12.isDisabled = false,
      S12.isOccupiedCW = false,
      S12.isOccupied = true,
      S12.isOccupiedCCW = false,
      S15.isDisabled = false,
      S15.isOccupiedCW = true,
      S15.isOccupied = false,
      S15.isOccupiedCCW = false,
      S12.GlobalState,
      T1.Straight,
      S15.GlobalState
    }
  }
}
```

Figure 4.18: An execution trace regarding Zone No. 1 in the MoDeS^3 safety-logic model, describing the consecutive states of the component if a train arrives to section *S12*.

Chapter 5

Implementation

This section introduces implementation details regarding the development of the Gamma framework. It starts with the introduction of the tools and frameworks Gamma builds upon. Next, the architecture of the framework is presented, which is followed by the introduction of the modeling tools integrated to the framework. Finally, the source code generator of Gamma is presented.

5.1 Technologies

We have 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 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[1] is an open-source, platform-independent integrated development environment (IDE). It consists of a base workspace (the basis of all Eclipse distributions) and a plug-in system. The 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 and we can install the Gamma framework to support the composition and verification of reactive systems.

Eclipse Modeling Framework Eclipse Modeling Framework[2] (EMF) is an Eclipse-based modeling framework with a strong support for code generation. 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 XML Metadata Interchange[3] (XMI) format, EMF provides design 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 support users in the modification and editing of their models.

[1] `https://eclipse.org`
[2] `http://eclipse.org/modeling/emf/`
[3] `https://www.omg.org/spec/XMI/`

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

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 textual grammar. 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, but each part of the metamodel of the Gamma language was created manually.

Xtend 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 both syntactically and semantically, but it offers a more compact 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 during the development of the Gamma framework. As a result, the codebase has remained relatively small while the source code itself has remained readable and concise.

5.1.3 VIATRA framework

VIATRA[6] is an Eclipse project that supports the development of model transformations with a large variety of tools. The model transformations of the Gamma framework heavily rely on VIATRA.

Most importantly, VIATRA offers a language that supports the definition of graph patterns over EMF models in a declarative way [47]. Since EMF models can be regarded as typed graphs where classes are nodes and their associations are edges, their transformations can be executed as graph transformations. With the declarative definition of graph patterns, VIATRA allows users to focus only on the types of elements, associations and the value of attributes, the retrieval of the corresponding elements is taken care of by VIATRA.

Furthermore, VIATRA supports the definition of model transformations using transformation rules. Each rule is based on a single graph pattern (declarative approach), which specifies the elements to be transformed (LHS). The RHS of the rule can be implemented

[4] http://eclipse.org/Xtext/
[5] http://eclipse.org/xtend/
[6] https://www.eclipse.org/viatra/documentation/

in Xtend (imperative approach). Also, the order in which the transformation rules are executed needs to be specified either with *1)* priorities or *2)* manual function calls in the imperative code.

5.2 Architecture

The architecture of Gamma is plug-in-based, which makes the framework modular, customizable, and easily extensible. The functionalities introduced in Chapter 3 are 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.

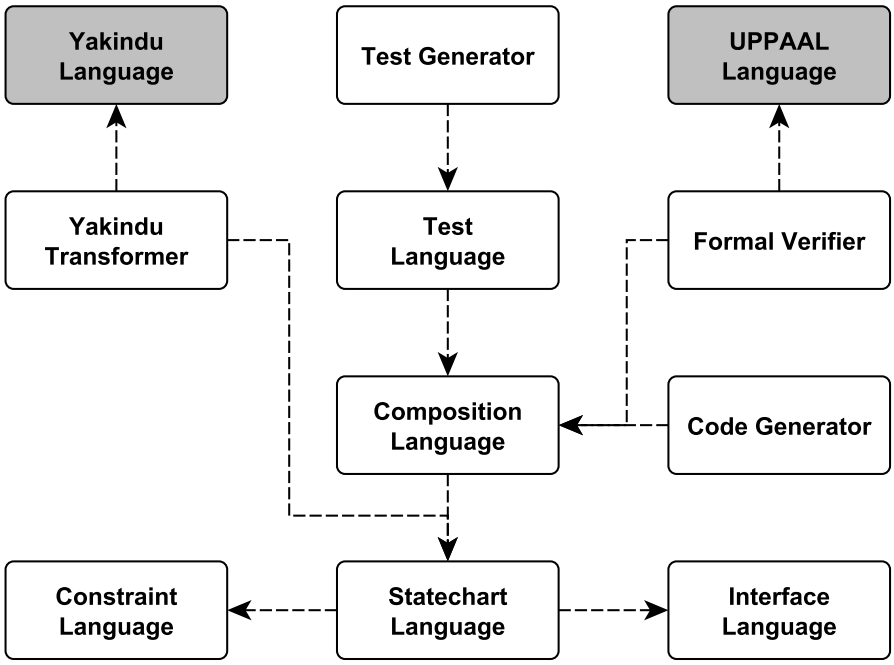

Figure 5.1: The plug-in dependencies of the framework. Plug-ins that are part of the core framework are depicted with white rectangles, whereas external dependencies are depicted with gray rectangles.

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 analysis languages can be introduced to the Gamma framework by defining the necessary model transformations and implementing them as a plug-in.

5.3 Integrated Modeling Languages

Currently, a single engineering and a single formal modeling language is integrated to the Gamma framework, Yakindu and UPPAAL, respectively. The integration of additional ones is supported by the plug-in architecture of Gamma, as presented in Section 5.2. Also, the integration of an additional model checking framework called Theta [48] is under development. This section introduces the features of modeling languages Yakindu

and UPPAAL, which heavily influenced the design and implementation techniques of the Gamma framework, as well as the principles of the Theta integration.

5.3.1 Integrated Engineering Language: Yakindu

YAKINDU Statechart Tools (SCT)[7] is a toolkit for the model-driven development of reactive 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. Interfaces, variables, triggers, guards and actions of transitions can be specified using a textual notation. A Yakindu statechart with basic model elements 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.

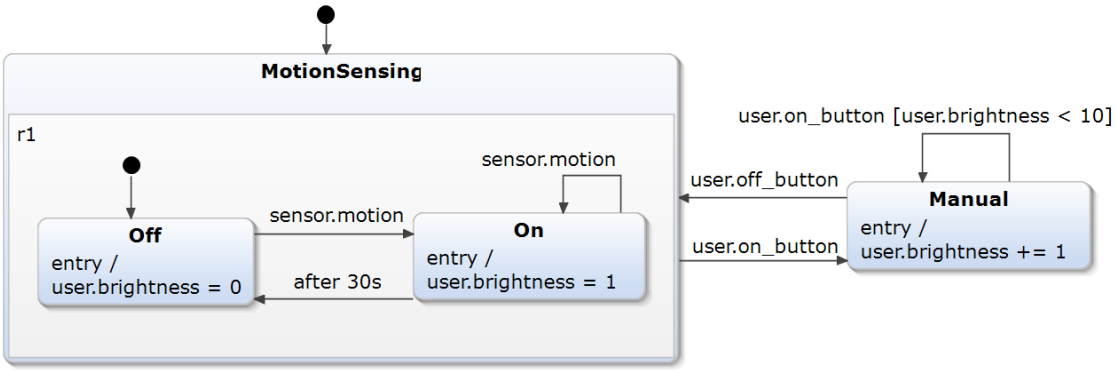

Figure 5.2: The graphical representation of a Yakindu statechart.

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.

[7] https://www.itemis.com/en/yakindu/state-machine/

5.3.2 Integrated Model Checker: UPPAAL

UPPAAL[8] is a software tool for the modeling, validation, simulation and formal verification of networks of timed automata. 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 [49, 50]. An UPPPAL automaton with synchronization channels is depicted in Figure 5.3.

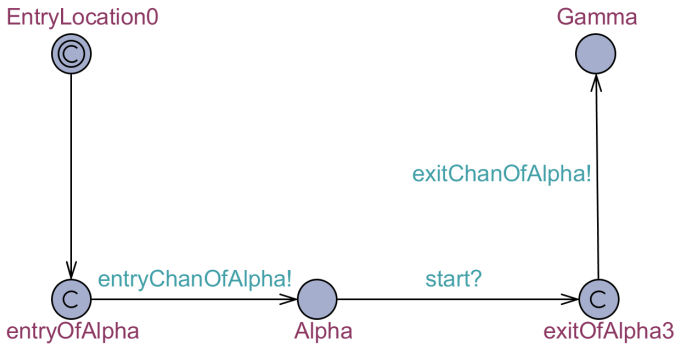

Figure 5.3: The graphical representation of an UPPAAL automaton.

UPPAAL is capable of performing formal verification on the defined timed automaton network using model checking techniques. Requirements on the systems' behavior can be described with temporal logic expressions. The language supported by UPPAAL is the subset of computation time logic (CTL) [51]. CTL is a branching-time logic which means its model of time is a tree-like structure. It starts from a root (the initial state) and each branch represents a possible execution sequence. The nodes of the branches represent the states the system assumes throughout the execution sequence.

An UPPAAL CTL expressions consists of a *path quantifier*, a *temporal operator* and a *state expression* [52]. The state expression can be any boolean expression that is valid in UPPAAL. UPPAAL does not support the combination of temporal operators, but a special class of expressions is supported by the "leads to" operator. The possible combination of path quantifiers and temporal operators are as follows.

- `A[]` ϕ: ϕ must hold in all states of all paths of the execution tree.
- `A<>` ϕ: ϕ must hold in at least one state of each path of the execution tree.
- `E[]` ϕ: ϕ must hold in all states of at least one path of the execution tree.
- `E<>` ϕ: ϕ must hold in at least one state of the execution tree.
- ϕ `-->` ψ: if ϕ occurs in state s, ψ must hold in at least one state of each path of the execution tree starting from state s. It is equivalent with the following expression: `A[](`ϕ `implies A<>` ψ`)`.

Figure 5.4 depicts the CTL expressions that are accepted by UPPAAL. The filled circles represent system states where the state expression ϕ holds.

[8] `http://www.uppaal.org/`

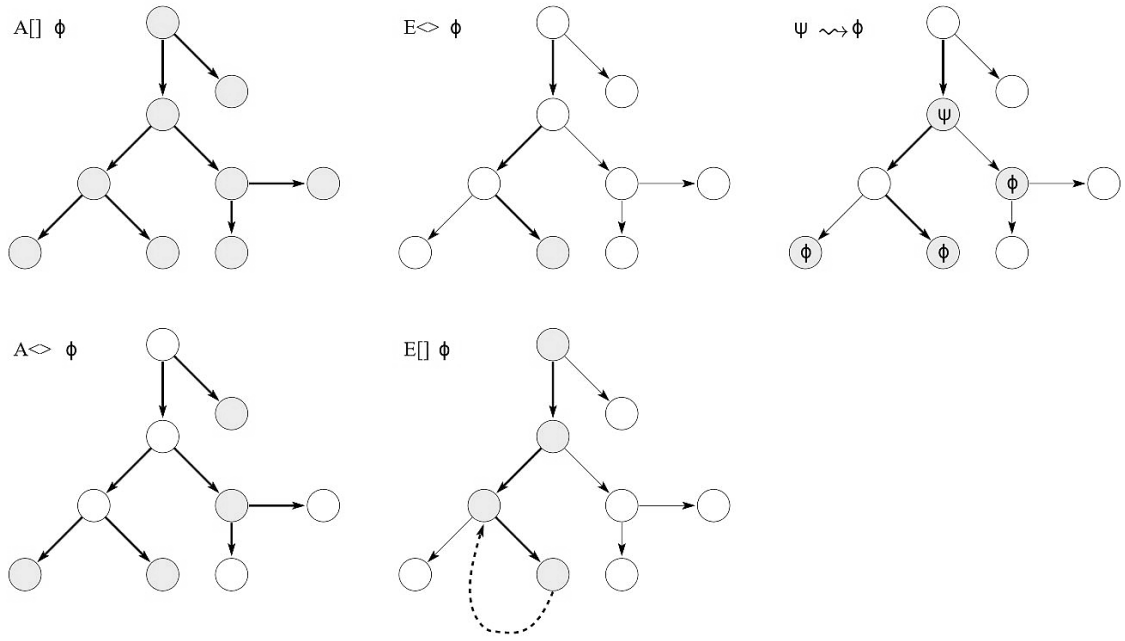

Figure 5.4: Temporal operators supported by UPPAAL.

Source: [52].

Supporting formal verification with the Gamma framework

As presented in Section 3.5, formal verification, back-annotation of the results and generation of a state-covering test-suite are supported with a GUI (see Figure 5.5). 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 selected Gamma models.

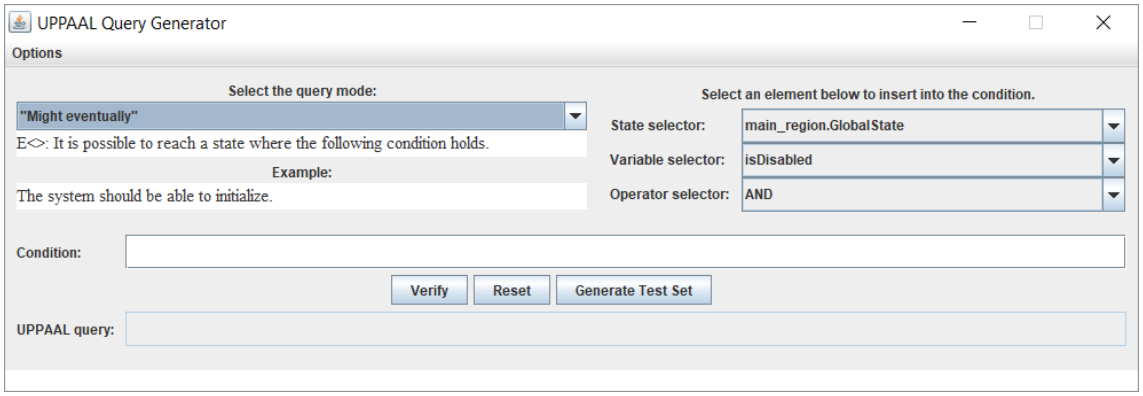

Figure 5.5: The window supporting the verification functionalities of the Gamma framework.

In the upper-left part of the window users can choose the temporal operator that specifies the states in which the formulated conditions must hold. Each temporal operator supported by UPPAAL can be selected. Also, these temporal operators are presented with examples. The selectors in the upper-right part can be used to formulate the desired condition using **States**, **Variables** and **Operators**. Selector **States** contains the states of

the model under verification, whereas selector **Variables** contains 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 case 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 an execution trace serving as proof or counter-example. Such execution traces are automatically back-annotated to the Gamma Test Language (see Section 4.8), so users can examine them in a familiar domain instead of the UPPAAL language. In addition to Gamma traces, JUnit test classes based on the UPPAAL traces are automatically generated during back-annotation.

Button **Generate Test Set** can be used to generate a state-covering test suite for the selected Gamma model. The automatic generation of transition-coverage test suite is not yet supported on the GUI.

5.4 Generated Source Code: Java

This section introduces the Java code generator of the Gamma framework in detail, which supports source code generation for composite models. The code generator is capable of generating functionally working code from interfaces, synchronous components (synchronous composite components and cascade composite components) and asynchronous components (asynchronous adapters and asynchronous composite components). However, code generation from statechart models is not supported, for this purpose, the code generator of Yakindu is used. Nevertheless, such generated statechart models are integrated to the Gamma code base using wrapper classes.

5.4.1 Interfaces

Every Gamma interface is transformed to a Java interface, each containing 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 *Provided* and *Required*, which describe contracts for listeners processing output events on ports implementing the interface in the respective mode. Figure 5.6 describes the Java interface that is generated from the Gamma interface *Train*, presented in the lower right of Figure 4.5.

Inner 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 *Train* has two `out` events, *occupy* and *unoccupy*, thus, a port realizing it in provided mode is able to dispatch *occupy* and *unoccupy* events, which is indicated by raising methods *raiseOccupy* and *raiseUnoccupy*. 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 realizing the particular Gamma interface in required mode. In this example, as Gamma interface *Train* does not contain any `in` events, *Listener.Required* is empty.

```
public interface TrainInterface {
  interface Provided extends Listener.Required {
    // Checking out-events
    public boolean isRaisedOccupy();
    public boolean isRaisedUnoccupy();
    // Handling registered listeners
    void registerListener(Listener.Provided listener);
    List<Listener.Provided> getRegisteredListeners();
  }
  interface Required extends Listener.Provided {
    // The raising methods of Listener.Provided and listener handlers
    void registerListener(Listener.Provided listener);
    List<Listener.Provided> getRegisteredListeners();
  }
  interface Listener {
    interface Provided {
      // Raising in-events
      void raiseOccupy();
      void raiseUnoccupy();
    }
    interface Required {
    }
  }
}
```

Figure 5.6: Java interfaces generated form interface *Train* in the safety-logic model of MoDeS^3.

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 (*isRaisedOccupy* and *isRaisedUnoccupy*), which can be used to check whether a certain event has been dispatched recently: the last cycle in case of synchronous components and the last cycle of wrapped synchronous components in case of asynchronous components. Both *Provided* and *Required* interfaces contain methods *registerListener* and *getRegisteredListeners*, which support the registration of listener objects. When a port dispatches a particular event, the corresponding "raise" methods of the registered listener objects are called. Interface *Required* is very similar to *Provided*; the difference is that it turns interfaces the other way around as described in Section 4.6, otherwise all its functionalities are analogous.

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

5.4.2 Components

Each Gamma component is transformed to a Java class. The generated Java classes are slightly different depending on their type, i.e., statechart, synchronous composite or asyn-

chronous composite. Moreover, a simple Java interface is generated for each component, through which the basic functionalities of the component, (initialization/reset, ports and in case of synchronous components, the initiation of a cycle) are reachable.

Synchronous components

Java classes generated from synchronous components can either represent an atomic component, that is, a statechart, or a composite component, that is, a synchronous composite component or a cascade composite component.

Atomic components Figure 5.7 and 5.8 describe the most important elements of the Java interface and Java class generated from Gamma statechart *Section* (presented in Figure 4.6). A Java class representing an atomic component has the following fields:

- a single statechart object implementing the behavior of the component (generated by integrated, external code generators or implemented manually),
- port objects representing Gamma ports, and
- a pair of queue implementations responsible for storing incoming events.

The queues are used as buffers of events before they are written into the cells of the statechart implementation (recall the event vector definition in Section 4.7.2). It is important to note that this is the only place where events are buffered, composite components containing these atomic components merely delegate the dispatch of events in accordance with the specified port bindings.

Furthermore, the queues are used in accordance with the semantics of the container composite component. If the container is a synchronous composite component, the contained components cannot affect each other in a single execution cycle, therefore, events under process (received in the previous cycle) and events received in the current execution cycle (to be processed in the next cycle) have to be separated and stored in different queues. Accordingly, the queues have to be swapped at the beginning of each execution cycle to facilitate the process of events received in the previous cycle and store new incoming events in the emptied queue. Nevertheless, only one queue is used in case of cascade composite components as the result of an event dispatch is observable immediately by the recipients in this case.

The interface of an atomic component provides access to the contained port instances, enables the reset and the execution of the component.

Composite components Figure 5.9 and 5.10 describe the most important elements of the Java interface and Java class generated from the Gamma synchronous composite component *Z1* (presented in Figure 4.12).

The fields represent contained component instances as well as ports used for communication with the environment (similarly to atomic components).

Method *initialize* is responsible for creating the channels, that is, it registers the corresponding ports of components as *listeners* using the *registerListener* method of ports. Furthermore, a getter methods is generated for each port instance. This way the registration of unique listeners is also supported, thus, users can be notified about the occurrence of certain events.

```
public interface SectionStatechartInterface {
  // Getting the ports
  ProtocolInterface.Required getProtocolInCCW();
  ProtocolInterface.Provided getProtocolOutCCW();
  ProtocolInterface.Required getProtocolInCW();
  ProtocolInterface.Provided getProtocolOutCW();
  SectionControlInterface.Provided getSectionControl();
  TrainInterface.Required getTrain();
  // Resetting the component
  void reset();
  // Initiating a cycle
  void runCycle();
}
```

Figure 5.7: Java interface generated for statechart definition *Section* in the safety-logic model of MoDeS^3.

As mentioned in Section 4.6.3, 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 mechanism is supported with a pair of queues in the atomic component implementation, but the swap of queues have to be controlled from the synchronous composite component implementation.

The synchronous composite 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 swaps the event queues of the contained components (*changeEventQueues*).
2. Calls method *runComponent*, which has the following effects.
   (a) It clears the output cells of contained atomic components indicating *transmission* of signals in the previous cycle. (Recall that output events are present for a single execution cycle only.)
   (b) It initiates a single cycle on each contained component.
   (c) It notifies any listeners registered to the system ports.

The *runCycle* method of **cascade composite components** are very similar. The only difference is that they do not have a *changeEventQueues* method as their components receive the events in the same cycle they have been dispatched. Furthermore, the contained components are executed (possibly multiple times) in the order as specified in the execution list.

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 *isEventQueueEmpty* method that checks whether any of the contained components has unprocessed events.

Asynchronous adapters

A Java class representing an asynchronous adapter has the following fields:

- a thread on which the adapter is executed,

```
public class SectionStatechart implements SectionStatechartInterface {
  // The wrapped statemachine implementation
  private SectionStatemachine sectionStatemachine =
    new SectionStatemachine();
  // Port instances
  private ProtocolInCW protocolInCW = new ProtocolInCW();
  private ProtocolInCCW protocolInCCW = new ProtocolInCCW();
  private ProtocolOutCCW protocolOutCCW = new ProtocolOutCCW();
  private ProtocolOutCW protocolOutCW = new ProtocolOutCW();
  private Train train = new Train();
  private SectionControl sectionControl = new SectionControl();
  // Event queues for the synchronization of statecharts
  private Queue<Event> eventQueue1 = new LinkedList<Event>();
  private Queue<Event> eventQueue2 = new LinkedList<Event>();
  ...

  /** Changes event queues and initiates a cycle run. */
  public void runCycle() {
    changeEventQueues();
    runComponent();
  }

  /** Initiates a cycle run without changing the event queues. */
  public void runCycle() {
    Queue<Event> eventQueue = getProcessQueue();
    while (!eventQueue.isEmpty()) {
      // Such event objects are created when an event on a port is raised
      Event event = eventQueue.remove();
      /* Events are identified by a string ID:
        <target-port-name>.<event-name> */
      switch (event.getEvent()) {
        // Writing the cells of the contained statechart implementation
        case "ProtocolInCW.Go":
          sectionStatemachine.getSCIProtocolInCW().raiseReserve();
        break;
        case "ProtocolInCW.Stop":
          sectionStatemachine.getSCIProtocolInCW().raiseRelease();
        break;
        // Writing of additional cells
        ...
      }
    }
    // Executing the contained statechart implementation
    sectionStatemachine.runCycle();
  }
}
```

Figure 5.8: Generated Java class wrapping statechart model *Section* in the safety-logic model of MoDeS^3.

```
public interface Z1Interface {
  // Getting the ports
  ProtocolInterface.Required getS12ProtocolInCCW();
  ProtocolInterface.Provided getS12ProtocolOutCCW();
  SectionControlInterface.Provided getS12Control();
  TrainInterface.Required getS12Train();
  // Resetting the component
  void reset();
  // Initiating cycles
  void runCycle();
  void runFullCycle();
}
```

Figure 5.9: Java interface generated for component *Z1* in the safety-logic model of MoDeS^3.

- an object representing the wrapped synchronous composite component,
- port objects,
- timer objects implementing clocks, and
- a *multiqueue* serving as a "bundle" for the separate messages queues.

The multiqueue is a third-party open-source concurrent collection implementation, which extends the existing Java concurrent collection library.[9] A multiqueue 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. Figure 5.11 describes the important elements of the Java class generated from the timed Gamma asynchronous adapter *Z1Adapter* (presented in Figure 4.15).

In the example, a single multiqueue is defined (*___asyncQueue*) which is instantiated with subqueue *clockMessages* (additional subqueues *protocolMessages* and *controlMessages* are not described in this example), representing the Gamma message queue. Note that the subqueue is instantiated with parameters *priority* and *capacity* as defined in the Gamma model.

As an asynchronous adapter 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 a 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),

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

```
public class Z1 implements Z1Interface {
  // Component instances
  private SectionStatechart S15 = new SectionStatechart();
  private TurnoutStatechart T1 = new TurnoutStatechart();
  private SectionStatechart S12 = new SectionStatechart();
  // Port instances
  private S12ProtocolInCCW s12ProtocolInCCW = new S12ProtocolInCCW();
  private S12ProtocolOutCCW s12ProtocolOutCCW = new S12ProtocolOutCCW();
  private S12Control s12Control = new S12Control();
  private S12Train s12Train = new S12Train();
  // Additional ports connected to T1 and S15...
  ...

/** Creates the channel mappings between the contained components. */
private void initialize() {
  // Registration of simple channels between T1 and S12
  T1.getProtocolInStraight().registerListener(S12.getProtocolOutCW());
  T1.getProtocolOutStraight().registerListener(S12.getProtocolInCW());
  S12.getProtocolInCW().registerListener(T1.getProtocolOutStraight());
  S12.getProtocolOutCW().registerListener(T1.getProtocolInStraight());
  // Channels between T1 and S15
  ...
}

/** Resets the contained components (recursively). */
public void reset() {
  S15.reset();
  T1.reset();
  S12.reset();
}

/** Initiates cycle runs until all event queues are empty. */
public void runFullCycle() {
  do {
    runCycle();
  }
  while (!isEventQueueEmpty());
}

/** Changes event queues and initiates a cycle run. */
public void runCycle() {
  // Changing the event queues for all synchronous subcomponents
  changeEventQueues();
  // Composite type-dependent behavior
  runComponent();
}

/** Initiates a cycle run without changing the event queues. */
private void runComponent() {
  // Starts with the clearing of the previous out-event flags
  clearPorts();
  // Running contained components
  S15.runComponent();
  T1.runComponent();
  S12.runComponent();
  // Notifying registered listeners
  notifyListeners();
}
```

Figure 5.10: Java class generated from component Z1 in the safety-logic model of MoDeS^3.

```
public class Z1Adapter implements Runnable, Z1AdapterInterface {
  // Thread running this adapter instance
  private Thread thread;
  // Wrapped synchronous instance
  private Z1 z1 = new Z1();
  // Port instances
  private S12ProtocolInCCW s12ProtocolInCCW = new S12ProtocolInCCW();
  private S12ProtocolOutCCW s12ProtocolOutCCW = new S12ProtocolOutCCW();
  ...
  // Clocks
  private ITimer timerService;
  private final int millisecondClock = 0;
  // Main queue
  private LinkedBlockingMultiQueue<String, Event> __asyncQueue =
    new LinkedBlockingMultiQueue<String, Event>();
  // Subqueues
  private LinkedBlockingMultiQueue<String, Event>.SubQueue clockMessages;
  ...

  private void init() {
    __asyncQueue.addSubQueue("clockMessages", 0, 8);
    clockMessages = __asyncQueue.getSubQueue("clockMessages");
    ...
    // Creating clock callback for the single timer service
    timerService.setTimer(createTimerCallback(), millisecondClock, 1, true);
    // Note: the thread has to be started manually
  }

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

  private void performControlActions(Event event) {
    /* Recall the ID structure of events: <target-port-name>.<event-name>.
       Additionally, asynchronous adapters can contain events of clocks,
       which are identified by the clock name. */
    String[] eventName = event.getEvent().split("\\.");
    // Clock trigger
    if (eventName.length == 1 && eventName[0].equals("millisecondClock")) {
      z1.runCycle();
      return;
    }
  }

  /** Starting the execution of the component on a separate thread. */
  public void start() {
    thread = new Thread(this);
    thread.start();
  }

}
```

Figure 5.11: Java class generated from the asynchronous adaptation of Zone No. 1 *Z1Adapter* in the safety-logic model of MoDeS^3.

a *full step*, or the *reset* of the wrapped component (see Definition 12). 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,
- port objects, and
- channel objects.

The generated Java class is simple, Figure 5.12 describes the important elements of the Java class generated from the Gamma asynchronous composite component *ModesTrack* (presented in Figure 4.16).

The functionality of an asynchronous composite component class is to bundle the contained components and forward incoming events from the system ports, and incoming events of channels to the corresponding ports of contained components. Note that this can be considered as flattening the asynchronous hierarchy to the asynchronous adapter 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.

Contrary to the previously presented component types, classes of asynchronous composite components connect the ports of contained components using specific channel objects. For each type of channel (defined by the interface of the connected ports) a simple Java class is generated, depicted in Figure 5.13. A channel class has a simple and clear interface. Its main goal is to provide a clear interaction point between asynchronous components (more specifically between their ports), even if they are not executed in the same process, but, for example, on different computational nodes. To provide inter-process communication, e.g., based on a DDS implementation, only these channel interfaces have to be realized along with the communicating agent, modification in the generated Java classes is not necessary.

```
public class ModesTrack implements ModesTrackInterface {
  // Component instances
  private Z1Adapter Z1 = new Z1Adapter();
  private Z2Adapter Z2 = new Z2Adapter();
  private Z3Adapter Z3 = new Z3Adapter();
  private Z4Adapter Z4 = new Z4Adapter();
  private Z5Adapter Z5 = new Z5Adapter();
  private Z6Adapter Z6 = new Z6Adapter();
  // Port instances
  private T1Turnout t1Turnout = new T1Turnout();
  ...
  private T6Turnout t6Turnout = new T6Turnout();
  private T1Train t1Train = new T1Train();
  ...
  private S31Train s31Train = new S31Train();
  private S01Control s01Control = new S01Control();
  ...
  private S31Control s31Control = new S31Control();
  // Channel instances
  private ProtocolChannelInterface channelS12ProtocolOutCCWOfZ1;
  private ProtocolChannelInterface channelS15ProtocolOutCWOfZ1;
  private ProtocolChannelInterface channelT1ProtocolOutDivergentOfZ1;
  ...

  private void init() {
    // Creating the channel objects
    channelS15ProtocolOutCWOfZ1 =
      new ProtocolChannel(Z1.getS15ProtocolOutCW());
    channelS15ProtocolOutCWOfZ1.registerPort(Z2.getS24ProtocolInCCW());
    channelS26ProtocolOutCCWOfZ3 =
      new ProtocolChannel(Z3.getS26ProtocolOutCCW());
    channelS26ProtocolOutCCWOfZ3.registerPort(Z6.getS27ProtocolInCW());
    channelS12ProtocolOutCCWOfZ1 =
      new ProtocolChannel(Z1.getS12ProtocolOutCCW());
    channelS12ProtocolOutCCWOfZ1.registerPort(Z4.getS01ProtocolInCW());
    channelS01ProtocolOutCWOfZ4 =
      new ProtocolChannel(Z4.getS01ProtocolOutCW());
    channelS01ProtocolOutCWOfZ4.registerPort(Z1.getS12ProtocolInCCW());
    channelT1ProtocolOutDivergentOfZ1 =
      new ProtocolChannel(Z1.getT1ProtocolOutDivergent());
    channelT1ProtocolOutDivergentOfZ1.registerPort(Z5.getS11ProtocolInCW());
    ...
  }

  ...

  public void start() {
    Z1.start();
    ...
    Z6.start();
  }

}
```

Figure 5.12: Java class generated from the asynchronous composite component *ModesTrack* in the safety-logic model of MoDeS^3.

```
public interface ProtocolChannelInterface {

  void registerPort(ProtocolInterface.Provided providedPort);
  void registerPort(ProtocolInterface.Required requiredPort);

}

public class ProtocolChannel implements ProtocolChannelInterface {
  // Single provided, possibly multiple required ports
  private ProtocolInterface.Provided providedPort;
  private List<ProtocolInterface.Required> requiredPorts =
    new LinkedList<ProtocolInterface.Required>();

  public ProtocolChannel(ProtocolInterface.Provided providedPort) {
    this.providedPort = providedPort;
  }

  public void registerPort(ProtocolInterface.Provided providedPort) {
    // Former port is forgotten
    this.providedPort = providedPort;
    // Registering the listeners
    for (ProtocolInterface.Required requiredPort : requiredPorts) {
      providedPort.registerListener(requiredPort);
      requiredPort.registerListener(providedPort);
    }
  }

  public void registerPort(ProtocolInterface.Required requiredPort) {
    requiredPorts.add(requiredPort);
    // Checking whether a provided port is already given
    if (providedPort != null) {
      providedPort.registerListener(requiredPort);
      requiredPort.registerListener(providedPort);
    }
  }

}
```

Figure 5.13: Java class generated from the channels connection *Protocol* port in asynchronous composite component *ModesTrack* in the safety-logic model of MoDeS^3.

Chapter 6

Case Study: MoDeS^3

This chapter demonstrates the usability of the Gamma framework by presenting a case study from the critical cyber-physical system domain. The case study is based on the MoDeS^3 project, already introduced in 4.1.

6.1 Introduction

The ultimate goal of the case study is to formally verify the correctness of the MoDeS^3 safely-logic using the Gamma framework, that is, trains cannot collide due to the erroneous design of the safety and communication protocol: *"Two separate trains must not be positioned on the same section."*

Generally, the main challenge for formal verification, and especially model checking is scalability, thus, it is very likely that the direct verification of a real railway system similar to the asynchronous MoDeS^3 model introduced in Section 4.6.4 is not (yet) feasible. Therefore, in this case study the safety-logic is redesigned and optimized in order to make the formal verification of the safety-logic realizable. During this process the following steps are taken.

- A simplified version of the MoDeS^3 track is introduced, which consists of eight section models forming a circle, with two trains on it.
- An iterative verification approach is executed on the simplified version of the MoDeS^3 track based on model decomposition (refinement) and the concept of bisimulation of models.

A bisimulation is a relation between transition systems. It associates systems that act in the same way, that is, one system simulates the other and vice versa [53]. In case of Gamma models it means that two models react to a particular series of external events received from the environment by transmitting the same series of events to the environment [54].

This case study focuses on the section model and track models built from it. To keep the case study simple, turnout track elements are not handled (as their large number of ports and events types would cause the proliferation of communication possibilities and states).

The rest of the chapter is structured as follows. Section 6.2 introduces the simplified MoDeS^3 track setup undergoing formal verification. Section 6.3 presents the bisimulation-based formal verification process. Finally, Section 6.4 summarizes the results of the case-study.

6.2 The Simplified MoDeS^3 Track Setup

The simplified MoDeS^3 track setup undergoing formal verification consists of two zones, each consisting of four track sections, that form a cycle. In the initial state, the setup contains two trains that are situated on opposite sections of the track, *S01* and *S05*, that is, there are three free sections in between them on both sides. The setup is depicted in Figure 6.1. Sections depicted in green are enabled sections occupied by trains, i.e., trains are permitted to move on them. Sections depicted in red are next to an occupied section and form the "aura" of the train. Auras are the means through which the safety protocol prevents collisions: the design of the logic will detect the collision of auras and stop the trains before a real collision occurs. The rest of the sections are blue.

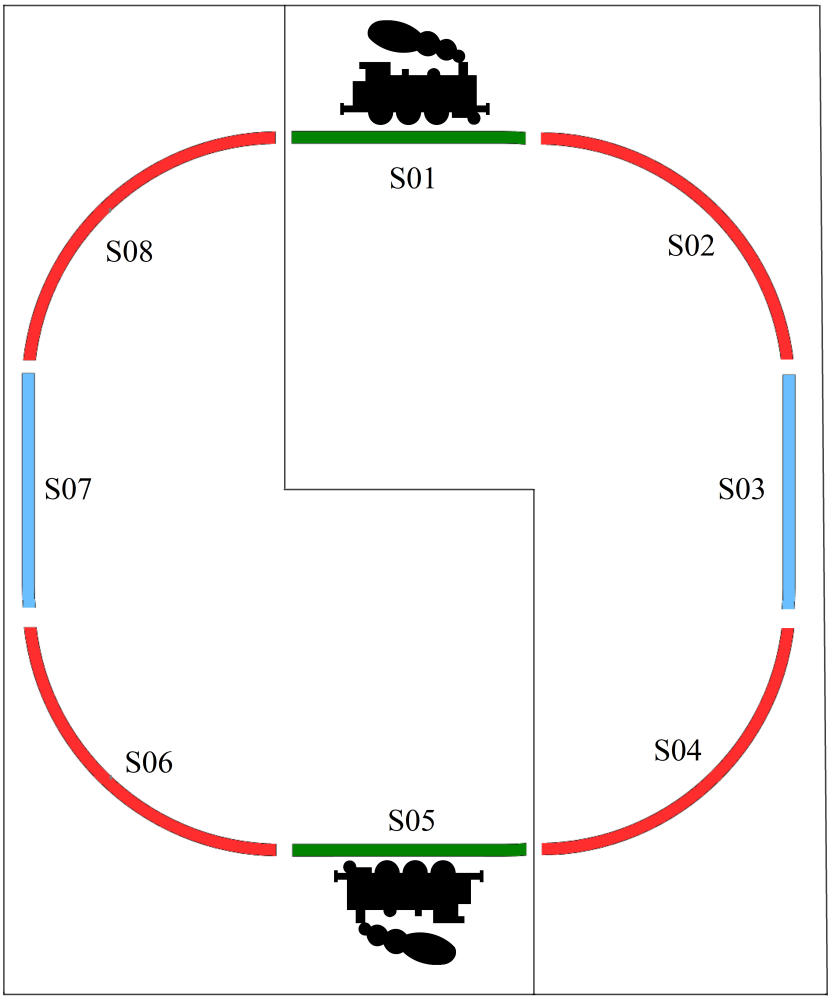

Figure 6.1: The simplified MoDeS^3 track setup undergoing formal verification.

6.3 Bisimulation-based Formal Verification

The bisimulation-based formal verification process of the simplified MoDeS^3 track consists of the following steps.

1. Interfaces and events supporting communication between track elements are defined.
2. A *high-level track model* is created, which defines the safety-logic satisfying the necessary safety-requirement: *"Two separate trains must not be positioned on the same section."* This model serves as the formal specification of the MoDeS^3 safety-logic in case of the simplified track.
3. The *high-level track model* is refined into a model called *medium-level track model* that contains two *zone* components (the type of which is called *medium-level zone model*), each representing four sections. The *medium-level zone model* serves as the formal specification of a zone controller. This refinement is a necessary design step, as the MoDeS^3 safety-logic is realized as the emergent behavior of multiple zone controllers. For the sake of simplicity, the synchronous composition mode is used between the zone controllers in the *medium-level track model* (opposed to the real MoDeS^3 track model depicted in Figure 4.16, which uses the asynchronous composition mode). Additionally, to facilitate the communication between the zone controllers, a new simpler communication protocol is designed.
4. The *medium-level zone model* created in the previous step is refined into a composite model containing four statechart models. The resulting model is called *low-level zone model*. Recurrently, this refinement step is necessary, as zone controllers consist of individual section components in the MoDeS^3 safety-logic. Additionally, a new section model is designed, which *1)* conforms to the new communication protocol, and *2)* is functionally equivalent to the section statechart model presented in [9] to support the use of the new model in the place of the old one without any possible compatibility issue. Finally, a *low-level track model* is created, which contains two *low-level zone model* components, thus, it can be considered as the low-level *physically deployable* realization of the MoDeS^3 safety-logic (in case of the simplified track).

Note that this is a refinement-based system design and verification technique frequently used during the design of safety-critical systems and supported by development processes, e.g., by the V-model [55].

The main goals, as depicted in Figure 6.2, are to formally prove that *1)* the *high-level track model* and the *medium-level track model* bisimulate each other, as well as, *2)* the *medium-level zone model* and the *low-level zone model* bisimulate each other. The existence of bisimulation relation *1)* would mean that the new communication protocol between zone controllers (*medium-level zone components* in the *medium-level track model*) is correct with respect to the formal specification of the MoDeS^3 safety-logic (*high-level track model*). Additionally, the existence of bisimulation relation *2)* would mean that a zone controller controlling four sections (*medium-level zone model*) can be correctly defined as the composition of four section statechart models (*low-level zone model*). If both bisimulation relation hold, then it means that the *high-level track model* can be correctly defined as the composition of two *low-level zone components* (the resulting model is the *low-level track model*), thus proving bisimulation relation *3)* on Figure 6.2 without ever analyzing the full low-level track model that is considerably more complex than the other two higher-level models. In this sense, results of the formal verification carried out on the *high-level track*

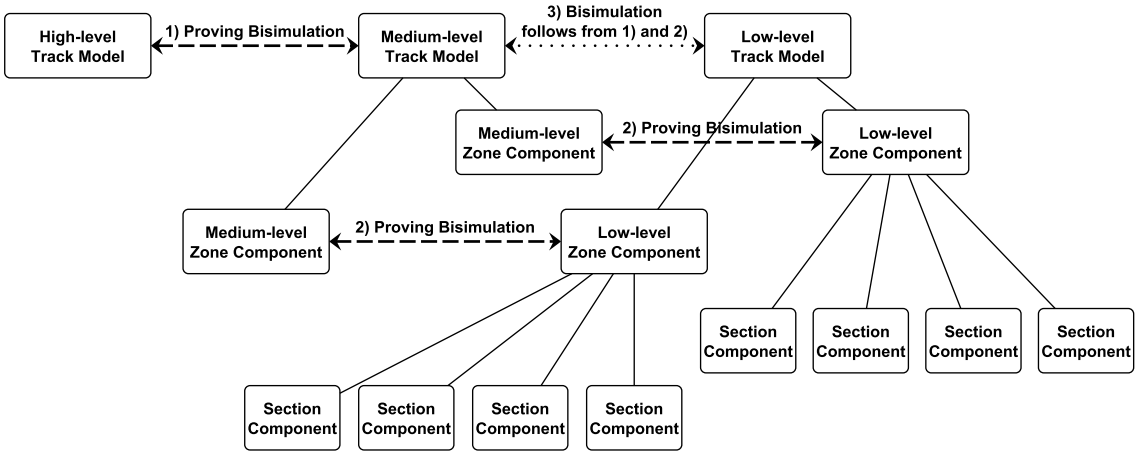

Figure 6.2: A schematic figure about the bisimulation-based formal verification process.

model would hold on the *low-level track model* too. This way, it would be sufficient to check the safety-requirements only on the simple, easily verifiable high-level track model, model checking on the complicated *low-level track model* would not be necessary.

An additional goal is to check whether track models created from the new and the old section models bisimulate each other. If so, then the verification results hold in case of the old safety-logic implementation as well.

Some of the models used in this case study, e.g., *high-level track model*, *medium-level zone model* (Section 6.3.2) and the train and oracle model (Section 6.3.4), are symmetrical models with many repetitive elements. Therefore, they are not created by hand, but generated on the basis of templates. These templates define the necessary model elements from a particular functional point of view, which are instantiated in each analogous part of the corresponding models during the generation process.

6.3.1 The Events of the New Communication Protocol

The new communication protocol is based on the interfaces and events presented in Figure 4.5.

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

- *Occupied*: this event is sent to both adjacent sections (neighbors) from a particular section that just got *occupied* by a train.
- *Unoccupied*: this event is sent to both adjacent sections (neighbors) from a particular section that just got *unoccupied* by a train.
- *Go*: this event is the *positive* answer to a previously sent occupied event. It means the train *can* proceed onto the particular section that sent this event.
- *Stop*: this event is the *negative* answer to a previously sent occupied event. It means the train *can not* proceed onto the particular section that sent this event.

Interface *Train* contains events that are used between section and train models and hold information about the arrival or the leaving of a train. The semantics of the *Train* events are as follows.

- *Occupy*: this event is sent to a section that just got *occupied* by a train.
- *Unoccupy*: this event is sent to a section if a train has *unoccupied* it.

Interface *SectionControl* contains events that are used between section and train models and permit or deny the moving of a train on a certain section. The semantics of the *SectionControl* events are as follows.

- *Enable*: this event is sent to a train by a section if it *permits* its moving, i.e., there is no dangerous situation, the train can proceed.
- *Disable*: this event is sent to a train by a section if it *denies* its moving, i.e., there is a dangerous situation, the train *can not* proceed.

Additionally, a new interface *TrainControl* is introduced, which is not actually a part of the real MoDeS^3 safety-logic, but is used in the verification process for the simulation of the motion of trains. The semantics of the *TrainControl* events are as follows.

- *MoveForward*: this event is sent to a train by the environment (user) if the train has to move *forward*.
- *MoveBackward*: this event is sent to a train by the environment (user) if the train has to move *backward*.

6.3.2 The Track Models

All track models have the same number and type of ports (same interface in the point of view of the environment): *Train* ports, on which the occupation and unoccupation events for particular sections are received, and *SectionControl* ports, on which the moving of trains are permitted or denied (see Figure 6.3).

```
...Track [
  port Train1 : requires Train,
  ...
  port Train8 : requires Train,
  port SectionControl1 : provides SectionControl,
  ...
  port SectionControl8 : provides SectionControl
] {
  ....
}
```

Figure 6.3: The ports of *high-level*, *medium-level* and *low-level track models* present in the MoDeS^3 verification process.

High-level Track Model

The *high-level track model*, presented in Figure 6.4 is a single statechart, with a single state, and two boolean variables for each section. One variable denotes whether a particular section is *occupied* by a train, whereas the other one denotes whether it is *disabled*. The transitions handle the situations when trains move on particular sections and disable/enable the sections when necessary in accordance with the safety-requirement. The explanation of behavior can be found in Figure 6.4 in the form of comments. Note that this model is symmetric, the transitions handling the actions of *S01*, *S02*, ..., *S08* are analogous.

Medium-level Track Model

The *medium-level track model* is a composite model that consists of two statecharts (*medium-level zone components*), each controlling four sections. This model refines the *high-level track model* by introducing *Protocol* ports in between the two halves of the track, that is, between *S08* and *S01*, and *S04* and *S05*, on which events of the new communication protocol are transmitted. Contrary to the *high-level track model*, communication between the zones is necessary to learn whether neighboring tracks on the edge of zones are occupied or not. Similarly to the *high-level track model*, this model is also symmetric: the endpoints of both *medium-level zone components* can be handled analogously. The *medium-level zone model* controlling four sections is presented in Figure 6.5, whereas the *medium-level track model* composing two such statechart components is depicted in Figure 6.6.

Low-level Track Model

The *low-level track model* is a composite model that consists of two *low-level zone components*. This model refines the *medium-level track model* by introducing *Protocol* ports in between *all* section components of the track. The *low-level zone model* composing four section models is depicted in Figure 6.7, whereas the *low-level track model* is depicted in Figure 6.8.

6.3.3 The New Section Model

The old section model, as presented in [9], behaves correctly with respect to the safety requirements. However, using the Gamma framework, the goal in this case study is to create a simpler model with the same behavior, in order to facilitate formal verification. During the redesign process, the most important aspect is to decrease the number of reachable states of the model. If the same behavior is realizable with a smaller number of states, the model checker will be able to explore the whole state space with much less resources. Furthermore, this model has to conform to the redesigned communication protocol. Figure 4.6 depicts all important transitions as well as static parts of the new section model, that is, ports, variables and states.

Ports The section model has four ports realizing the *Protocol* interface, two for both endpoints; one for transmitting the protocol events and one for receiving them. The endpoints are called *CW* and *CCW*, which stand for *clockwise* and *counter-clockwise*, considering the fact that the sections are connected to form a circle. Moreover, the model

has a port realizing the *SectionControl* interface for permitting/denying the moving of a train, and another port realizing the *Train* interface for being able to notice the arrival and leaving of trains.

Variables The section model has four boolean variables: *isDisabled*, *isOccupiedCW*, *isOccupied* and *isOccupiedCCW*. Table 6.1 summarizes the semantics of the variables.

Table 6.1: The semantics of the boolean variables in the new MoDeS^3 section model.

Variable	False	True
isDisabled	The section is enabled. It permits the passing of trains.	The section is disabled. Trains cannot move on it.
isOccupiedCW	The section connected to the CW endpoint is *not* occupied.	The section connected to the CW endpoint is occupied.
isOccupied	The section is *not* occupied.	The section is occupied.
isOccupiedCCW	The section connected to the CCW endpoint is *not* occupied.	The section connected to the CCW endpoint is occupied.

States The section model has three explicit states: *Stable*, *WaitForCW* and *WaitForCCW*. State *Stable* represent a stable state in the section model. In this state all variable values are considered valid, the statechart does not wait for any events from the neighbors or the environment. However, *WaitForCW* and *WaitForCCW* represent temporary states. In state *WaitForCW*, either a *go* or *stop* event is expected in response to a transmitted *occupied* event from the *CW* neighbor, and in *WaitForCCW* from the *CCW* neighbor.

By introducing only three explicit states in the section model and relying on two boolean variables to store information about the environment of the section we strive to create a reduced, easily verifiable statechart model with simple communication with its neighbors that can be easily described as a transition system.

Transitions The transitions of the model are defined in Table 6.2 with the following attributes: ID, source state, trigger, guard, action and target state. In case of triggers, guards and actions, the syntax introduced in Section 4.5 is used. In the table the names of states and ports are abbreviated, only the capital letters of a particular port or state are used for identification, e.g., *ProtocolInCW* is abbreviated to *PICW*. Furthermore, state *WaitForCCW* is abbreviated to *CCW*. For the sake of compactness only the transitions reacting to the events of the *CW* neighbor are presented. However, note that this is a symmetric model. Accordingly, analogous transitions are also defined for the events of the *CCW* neighbor.

The explanations with respect to transitions in state *Stable* are as follows.

- Trigger *occupy*: a train occupies the section. In every case, the section becomes occupied and neighbors are notified.
  1. If the section and both neighbors are free, the train has just appeared (initialization).
  2. If CW neighbor is occupied, the train arrives from there.
  3. If CCW neighbor is occupied, the train arrives from there.

4. If both neighbors are occupied, there is a dangerous situation so section gets disabled.

- Trigger *occupied*: neighbor becomes occupied. In every case, neighbor occupation is updated.

5. If other neighbor is occupied, auras collide, sending *stop*.
6. Otherwise good to go, sending *go*.

- Trigger *unoccupied*: neighbor becomes unoccupied. In every case, neighbor occupation is updated.

7. If other neighbor is occupied (and thus, disabled), sending *go* to enable it.
8. If the section is disabled, it remains disabled, sending *disable*.
9. Otherwise no further action.

- Trigger *stop*: neighbor sends *stop*.

10. Dangerous situation, disabling section.

- Trigger *go*: neighbor sends *go* and no neighbors send *stop* at the same time.

11. If section is disabled, notification about occupation is repeated and state *WaitForCCW* is assumed.
12. Otherwise confirming that the section is enabled.

The explanations with respect to transitions in state *WaitForCCW* are as follows.

13. Trigger *go*: neighbor sends *go* as response and no neighbors send *stop* at the same time. The section is disabled at this time, therefore, it must be enabled.
14. Trigger *stop*: neighbor sends *stop* as response. The section is disabled at this time, no further action.

6.3.4 Proving the Bisimulation Relations

To prove the bisimulation relation between the corresponding models, a train model, an oracle model and two bisimulation system models have to be created.

The Train Model

A train model is needed in the verification process to simulate the motion of the trains correctly. In this case study it is worth creating a compound model that is able to

- handle and simulate two trains on the track,
- simulate the motion of a particular train on *two separate track instances* (e.g., *high-level track model* and *medium-level track model*) at the same time, which is crucial for the proving of bisimulation, and
- differentiate eight section elements in the underlying track models.

Table 6.2: The transitions of the new MoDeS^3 section model.

ID	Source	Trigger	Guard	Action	Target
1	S	T.occupy	not isOccupiedCW and not isOccupied and not isOccupiedCCW	isOccupied = true; raise POCW.occupied; raise POCCW.occupied	S
2	S	T.occupy	isOccupiedCW and not isOccupied and not isOccupiedCCW	isOccupied = true; raise POCW.occupied; raise POCCW.occupied	S
3	S	T.occupy	not isOccupiedCW and not isOccupied and isOccupiedCCW	isOccupied = true; raise POCW.occupied; raise POCCW.occupied	S
4	S	T.occupy	isOccupiedCW and not isOccupied and isOccupiedCCW	isOccupied = true; isDisabled = true; raise SC.disable; raise POCW.occupied; raise POCCW.occupied	S
5	S	PICW.occupied	isOccupiedCCW or isDisabled	isOccupiedCW = true; raise POCW.stop	S
6	S	PICW.occupied	not (isOccupiedCCW or isDisabled)	isOccupiedCW = true; raise POCW.go	S
7	S	PICW.unoccupied	isOccupiedCCW and not isOccupied	isOccupiedCW = false; raise POCCW.go	S
8	S	PICW.unoccupied	isDisabled	isOccupiedCW = false; raise SC.disable	S
9	S	PICW.unoccupied	not (isDisabled or isOccupiedCCW and not isOccupied)	isOccupiedCW = false	S
10	S	PICW.stop	true	isDisabled = true; raise SC.disable	S
11	S	PICW.go && !(PICCW.stop || PICW.stop)	isDisabled	raise POCCW.occupied	CCW
12	S	PICW.go && !(PICCW.stop || PICW.stop)	not isDisabled	raise SC.enable	S
13	CCW	PICW.go && !(PICCW.stop || PICW.stop)	isDisabled	isDisabled = false; raise SC.enable	S
14	CCW	PICW.stop	isDisabled	No action	S

Figure 6.9 describes the train simulator model used for verification. This model is also symmetrical in many ways, as the two trains behave the same way, and a train behaves analogously on the first, second, etc. sections, when moving forward or backward.

As a physical system is modeled here (motion of the trains on the track), it is important to keep in mind that the train moves "gradually" onto a section, and also, leaves it "gradually". For example, if a train is situated on *S01* and is moved onto an adjacent section *S02*, it is done so in the following steps (let us assume no dangerous situation is present).

1. The train is situated on *S01* and receives a *moveForward* event.
2. The train moves onto *S02*, but does not leave *S01*. The trains notifies *S02* about its arrival. At this time, the train is situated on both sections.
3. The train receives another *moveForward* event.
4. The train moves onto *S02* leaving *S01*. The trains notifies *S01* about its leaving. At this time, the train is situated only on section *S02*.

These steps are defined in Figure 6.9 by the last three transitions. Note that trains can move only if they are not disabled, that is, their *isDisabled* variable is not set to true. Furthermore, it is worth noting how the position of a train is coded. If variable *position* is set to a number with a single digit, e.g., 1, it means the train is situated only on a single section, e.g., *S01*. If the position is set to a number with two digits, e.g., 12, it means the trains is situated on two sections, e.g., *S01* and *S02* at the same time. A single train can be situated on at most two sections.

Oracle model

To prove that the high-level, medium-level and low-level MoDeS^3 track models bisimulate each other, an oracle statechart model is needed that checks the section control events (*enable* or *disable*, received through *HigherLevelSectionControl* and *LowerLevelSectionControl* ports) of each section model, and compare them. The comparison of events is realized using complex triggers. In case of a difference, the oracle raises an *error* event on port *Error* and goes to the *Error* state. The oracle model is depicted in Figure 6.10.

Bisimulation system model

The proof of the corresponding track and zone models bisimulating each other is given by executing formal verification on two variants of the *BisimulationSystem* cascade composite Gamma model, which composes the train model, two track or zone models (a higher- and a lower-level model in both cases) and an oracle model into a single system (depicted in Figure 6.11 and defined in Figure 6.12). The train model receives commands from the environment about moving the trains forward or backward, and based on those events, it notifies the track/zone models about the occupation and unoccupation of sections. Based on those events track/zone models permit or deny the moving of trains on certain sections by sending the necessary *enable* or *disable* events. These events are received by both *1)* the train model, which can set the *disable* flag of the trains based on those events (a single *disable* event from either track/zone model is enough to disable the train), and *2)* the oracle model, which compares the received events and raise an *error* event if inconsistency between the track models are found.

Note that the *BisimulationSystem* model is a *cascade* composite component, the characteristics of which is essential to execute all contained components in a specified order, while enabling their communication in a single turn.

The track-based and zone-based variants of the *BisimulationSystem* model are almost the same. The difference is that one compares the *high-* and *medium-level track models* (trains can move around on the track on eight adjacent sections as depicted in Figure 6.1), while the other one compares the *medium-* and *low-level zone models* (trains can move forward and backward on four adjacent sections as depicted in Figure 6.13).

Verifying the bisimulation relation between the track and zone models

The bisimulation relation between the track and zone models can be proven with model-checking. Both variants of the *BisimulationSystem* model have to be transformed to UPPAAL, and the following query has to be evaluated on them.

A[] !(P_mainRegionOfStatechartOforacle.Error)

This query is the formalization of the following statement:

"In the Bisimulation system, state Error of component oracle is never assumed."

If state *Error* of component oracle is never assumed, it is impossible for the two track/zone models in the bisimulation system to produce inconsistent section control events, since that inconsistency would trigger the transitions leading to the *Error* state in the oracle model. In other words, the bisimulation relation between the models holds.

By following the aforementioned instructions, we have formally proven with the help of the UPPAAL model-checker that the aforementioned condition *holds* on both variants of the bisimulation system model, that is, the presented *high-* and *medium-level track mode*l, and the *medium-* and *low-level zone models* bisimulate each other.

Verifying correspondence between the new and old section models

As section components are not used individually, but in composition with other section components, it is worth proving the bisimulation relation between track models (one consisting of the new and the other of old section models), and not between individual section models. Accordingly, a zone model has to be created using the old section model. This model can be created on the basis of the *low-level zone model* (Figure 6.7). The differences are the followings.

- The old section model must be used instead of the new one as the type of the components.
- The *Protocol* ports connected in the channels do not have to be changed, but it is worth noting that they realize a different interface. Note that the two models have the same ports apart from the *Protocol* ports.

Recurrently, the bisimulation relation between the *track models* consisting of the new and the old section models can be proven with the *BisimulationSystem* model. Either of the *high-*, *medium-* or *low-level track models* could be used in pair with the newly created model as their bisimulation relation has been proven. Nevertheless, to facilitate model checking, it is worth choosing the *high-level track model* because it is the simplest. Again,

this version of the *BisimulationSystem* model also has to be transformed to UPPAAL, and the following query has to be evaluated on it.

A[] !(P_mainRegionOfStatechartOforacle.Error)

By following the aforementioned instructions, we have formally proven that the condition *holds* on this variant of the bisimulation system model, that is, track models based on the new section model and the old section model bisimulate each other.

6.3.5 Formal Verification of the MoDeS^3 Safety-Logic

As the bisimulation relation between the *high-* and *medium-level track model*, and the *medium-* and *low-level zone models* are formally proven, the verification of any property regarding the section control events is valid for all track models, regardless on which of them the verification is performed. Naturally, the *high-level track model* is the simplest, thus, verification on this level is the fastest. Therefore, it is worth using the *high-level zone model* for the verification of the safety-requirement. To achieve this, a new Gamma model has to be created on which the verification can be carried out. As depicted in Figure 6.14, the model is the subset of the *BisimulationSystem* model, keeping the train and the high-level models while omitting the other track model and the oracle.

Similarly to the bisimulation relation, the standing of the safety-requirement in the MoDeS^3 track model, that is, *"Two separate trains must not be positioned on the same section."*, can be proven with model checking. The *HighLevelSystem* model has to be transformed to UPPAAL, and a formalized query has to be evaluated on it. In case of this model, there is no clear erroneous state the reachability of which can be verified. Instead, the following statement is formalized:

"In the HighLevelSystem system, a state, when two trains occupy the same section, is never assumed."

This statement can be formalized as follows, using the variables in the train simulator model:

A[] !((position1Oftrain == 81 || position1Oftrain == 1 || position1Oftrain == 12) && (position2Oftrain == 81 || position2Oftrain == 1 || position2Oftrain == 12)) &&
!((position1Oftrain == 12 || position1Oftrain == 2 || position1Oftrain == 23) && (position2Oftrain == 12 || position2Oftrain == 2 || position2Oftrain == 23)) &&
... &&
!((position1Oftrain == 78 || position1Oftrain == 8 || position1Oftrain == 81) && (position2Oftrain == 78 || position2Oftrain == 8 || position2Oftrain == 81))

Recall the mechanism according to which the states of the trains are encoded in the train simulator model (Section 6.3.4). Note that each section is checked one by one for both trains.

By following the aforementioned instructions, we have formally proven with the help of the UPPAAL model-checker that the aforementioned condition *holds* on the high-level system model, that is, two trains can never occupy the same section. Due to the results presented in Section 6.3.4, this condition also holds in the *low-level track model* and in the track model based on the old section models.

6.4 Summary

In this case study the features of the Gamma framework has been utilized to redesign and formally verify the MoDeS^3 safety-logic. During this process

1. a new statechart model and a new communication protocol have been designed for the section track element,
2. a simplified version of the MoDeS^3 track model has been introduced as the basis of the verification,
3. an iterative verification approach has been executed based on model decomposition and the concept of bisimulation. For this process, additional models have been introduced.

The verification result have shown that *1)* the new section and the old sections bisimulate each other, and *2)* the new communication protocol of the section model satisfies the necessary safety-requirement. As a result of the formal semantics of the Gamma Composition Language to which both formal verification transformations and code generation conform, the source code generated from the composition models and deployed on the physical track model must also work correctly.

```
statechart HighLevelTrackModel [
  // Train and SectionControl ports
  ...
] {
  // Variables denoting whether a section is occupied by a train
  var isOccupied1 : boolean := false
  ...
  var isOccupied8 : boolean := false
  // Variables denoting whether a particular section is disabled
  var isDisabled1 : boolean := false
  ...
  var isDisabled8 : boolean := false
  // Single state in the main region
  region mainRegion {
    initial Initial
    state Stable
  }
  transition from Initial to Stable
  // A train emerges on S01
  transition from Stable to Stable when Train1.occupy
    [not (isOccupied7 or isOccupied8 or isOccupied2 or isOccupied3)] /
    isOccupied1 := true; raise SectionControl1.enableSection
  transition from Stable to Stable when Train1.occupy
    [isOccupied8 and not (isOccupied1 or isOccupied2 or isOccupied3)] /
    isOccupied1 := true; raise SectionControl1.enableSection
  transition from Stable to Stable when Train1.occupy
    [isOccupied8 and (isOccupied1 or isOccupied2 or isOccupied3)] /
    isOccupied1 := true; isDisabled1 := true;
    raise SectionControl1.disableSection
  transition from Stable to Stable when Train1.occupy
    [isOccupied2 and not (isOccupied1 or isOccupied8 or isOccupied7)] /
    isOccupied1 := true; raise SectionControl1.enableSection
  transition from Stable to Stable when Train1.occupy
    [isOccupied2 and (isOccupied1 or isOccupied8 or isOccupied7)] /
    isOccupied1 := true; isDisabled1 := true;
    raise SectionControl1.disableSection
  // A train leaves S01
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled2] /
    isOccupied1 := false; isDisabled1 := false;
    raise SectionControl2.disableSection
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled8] /
    isOccupied1 := false; isDisabled1 := false;
    raise SectionControl8.disableSection
  // Enable certain sections if they disabled
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled3 and not (isDisabled2 or isDisabled8)] /
    isOccupied1 := false; isDisabled3 := false;
    raise SectionControl3.enableSection
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled7 and not (isDisabled2 or isDisabled8)] /
    isOccupied1 := false; isDisabled7 := false;
    raise SectionControl7.enableSection
  // A train simply left S01
  transition from Stable to Stable when Train1.unoccupy
    [not (isDisabled2 or isDisabled3 or isDisabled8 or isDisabled7)] /
    isOccupied1 := false
  ...
}
```

Figure 6.4: The *high-level track model* in the MoDeS^3 verification process.

```
statechart MediumLevelZoneModel [
  // Train and SectionControl ports
  ...
  // Protocol ports
  port ProtocolInCCW : requires Protocol,
  port ProtocolInCW : requires Protocol,
  port ProtocolOutCCW : provides Protocol,
  port ProtocolOutCW : provides Protocol
] {
  // Variables denoting if section is occupied or disabled
  ...
  // Variables denoting if next section in an adjacent component is occupied
  var isCCWOccupied : boolean := false
  var isCWOccupied : boolean := false
  region mainRegion {
    initial Initial
    state Stable
  }
  transition from Initial to Stable
  // A train emerges on S01
  transition from Stable to Stable when Train1.occupy
    [not (isCCWOccupied or isOccupied1 or isOccupied2)] /
    isOccupied1 := true; isDisabled1 := false;
    raise SectionControl1.enableSection; raise ProtocolOutCCW.occupied
  transition from Stable to Stable when Train1.occupy
    [isCCWOccupied and not (isOccupied1 or isOccupied2 or isOccupied3)] /
    isOccupied1 := true; raise SectionControl1.enableSection;
    raise ProtocolOutCCW.occupied
  transition from Stable to Stable when Train1.occupy
    [isCCWOccupied and (isOccupied1 or isOccupied2 or isOccupied3)] /
    isOccupied1 := true; isDisabled1 := true; raise SectionControl1.
    disableSection; raise ProtocolOutCCW.occupied
  transition from Stable to Stable when Train1.occupy
    [isOccupied2 and not (isOccupied1 or isCCWOccupied)] /
    isOccupied1 := true; raise ProtocolOutCCW.occupied
  // Protocol message from adjacent component
  transition from Stable to Stable when ProtocolInCCW.stop /
    isDisabled1 := true; raise SectionControl1.disableSection
  // A train leaves S01, keep S02 disabled if it was previously disabled
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled2] /
    isOccupied1 := false; isDisabled1 := false;
    raise SectionControl2.disableSection; raise ProtocolOutCCW.unoccupied
  // Enable S03 if it is disabled
  transition from Stable to Stable when Train1.unoccupy
    [isDisabled3 and not (isDisabled2)] /
    isOccupied1 := false; isDisabled3 := false;
    raise SectionControl3.enableSection; raise ProtocolOutCCW.unoccupied
  // A train simply left S01
  transition from Stable to Stable when Train1.unoccupy
    [not (isDisabled3 or isDisabled2)] /
    isOccupied1 := false; raise ProtocolOutCCW.unoccupied
  ...
```

Figure 6.5: The *medium-level zone model* in the MoDeS^3 verification process.

```
sync MediumLevelTrackModel [
  // Train and SectionControl ports
  ...
] {
  // Components
  component ccwFourSections : MediumLevelZoneModel
  component cwFourSections : MediumLevelZoneModel
  // Bindings
  bind Train1 -> ccwFourSections.Train1
  bind Train2 -> ccwFourSections.Train2
    ...
  bind Train7 -> cwFourSections.Train3
  bind Train8 -> cwFourSections.Train4
  bind SectionControl1 -> ccwFourSections.SectionControl1
  bind SectionControl2 -> ccwFourSections.SectionControl2
    ...
  bind SectionControl7 -> cwFourSections.SectionControl3
  bind SectionControl8 -> cwFourSections.SectionControl4
  // Channels connecting the two zone components
  channel [ cwFourSections.ProtocolOutCCW ] -o)-
    [ ccwFourSections.ProtocolInCW ]
  channel [ ccwFourSections.ProtocolOutCCW ] -o)-
    [ cwFourSections.ProtocolInCW ]
  channel [ cwFourSections.ProtocolOutCW ] -o)-
    [ ccwFourSections.ProtocolInCCW ]
  channel [ ccwFourSections.ProtocolOutCW ] -o)-
    [ cwFourSections.ProtocolInCCW ]
}
```

Figure 6.6: The *medium-level track model* in the MoDeS^3 verification process.

```
statechart LowLevelZoneModel [
  // Train and SectionControl ports
  ...
  // Protocol ports
  port ProtocolInCCW : requires Protocol,
  port ProtocolInCW : requires Protocol,
  port ProtocolOutCCW : provides Protocol,
  port ProtocolOutCW : provides Protocol
] {
  // Four section statecharts
  component section1 : SectionStatechart
  component section2 : SectionStatechart
  component section3 : SectionStatechart
  component section4 : SectionStatechart
  // Bindings
  bind ProtocolInCCW -> section1.ProtocolInCCW
  ...
  bind ProtocolOutCW -> section4.ProtocolOutCW
  bind Train1 -> section1.Train
  ...
  bind Train4 -> section4.Train
  bind SectionControl1 -> section1.SectionControl
  ...
  bind SectionControl4 -> section4.SectionControl
  // Channels connecting the four statecharts
  channel [ section2.ProtocolOutCCW ] -o)- [ section1.ProtocolInCW ]
  channel [ section1.ProtocolOutCW ] -o)- [ section2.ProtocolInCCW ]
  channel [ section3.ProtocolOutCCW ] -o)- [ section2.ProtocolInCW ]
  channel [ section2.ProtocolOutCW ] -o)- [ section3.ProtocolInCCW ]
  channel [ section4.ProtocolOutCCW ] -o)- [ section3.ProtocolInCW ]
  channel [ section3.ProtocolOutCW ] -o)- [ section4.ProtocolInCCW ]
}
```

Figure 6.7: The *low-level zone model* in the MoDeS^3 verification process.

```
sync LowLevelTrackModel [
  // Train and SectionControl ports
  ...
] {
  // Components
  component ccwFourSections : LowLevelZoneModel
  component cwFourSections : LowLevelZoneModel
  // Bindings
  bind Train1 -> ccwFourSections.Train1
  bind Train2 -> ccwFourSections.Train2
  ...
  bind Train7 -> cwFourSections.Train3
  bind Train8 -> cwFourSections.Train4
  bind SectionControl1 -> ccwFourSections.SectionControl1
  bind SectionControl2 -> ccwFourSections.SectionControl2
  ...
  bind SectionControl7 -> cwFourSections.SectionControl3
  bind SectionControl8 -> cwFourSections.SectionControl4
  // Channels connecting the two zone components
  channel [ cwFourSections.ProtocolOutCCW ] -o)-
    [ ccwFourSections.ProtocolInCW ]
  channel [ ccwFourSections.ProtocolOutCCW ] -o)-
    [ cwFourSections.ProtocolInCW ]
  channel [ cwFourSections.ProtocolOutCW ] -o)-
    [ ccwFourSections.ProtocolInCCW ]
  channel [ ccwFourSections.ProtocolOutCW ] -o)-
    [ cwFourSections.ProtocolInCCW ]
}
```

Figure 6.8: The *low-level track model* in the MoDeS^3 verification process.

```
statechart TrainSimulator [
  // Moving the trains forward and backward
  port TrainControl1 : requires TrainControl,
  port TrainControl2 : requires TrainControl,
  // Handling permissions and denials of sections
  // At the same time two tracks (lower-level and higher-level) are handled
  port HigherLevelSectionControl1 : requires SectionControl,
  port LowerLevelSectionControl1 : requires SectionControl,
  ...
  port HigherLevelSectionControl8 : requires SectionControl,
  port LowerLevelSectionControl8 : requires SectionControl,
  // Notifying sections about arrival and leaving
  // At the same time multiple tracks can be handled due to broadcast ports
  port Train1 : provides Train,
  ...
  port Train18 : provides Train
] {
  // If a train is disabled, it cannot move
  var isDisabled1 : boolean
  var isDisabled2 : boolean
  // The position of the trains
  var position1 : integer := 1
  var position2 : integer := 5
  region mainRegion {
    initial Initial
    state Stable
    choice ForwardChoice1
    // Additional choices for forward and backward motion
    ...
  }
  // Train1 disabled when moving onto the S01 from S02
  transition from Stable to Stable when
    HigherLevelSectionControl1.disableSection ||
    LowerLevelSectionControl1.disableSection [position1 = 12] /
    isDisabled1 := true; position1 := 1; raise Train2.unoccupy
  // Enabling a disabled train on S01
  transition from Stable to Stable when
    HigherLevelSectionControl1.enableSection &&
    !(HigherLevelSectionControl1.disableSection)
    [isDisabled1 and position1 = 1] / isDisabled1 := false
  transition from Stable to Stable when
    LowerLevelSectionControl1.enableSection &&
    !(LowLevelSectionControl1.disableSection)
    [isDisabled1 and position1 = 1] / isDisabled1 := false
  // Train1 moving forward
  transition from Stable to ForwardChoice1 when TrainControl1.moveForward &&
    !(HigherLevelSectionControl1.disableSection &&
    LowerLevelSectionControl1.disableSection ||
    ...
    HigherLevelSectionControl8.disableSection &&
    LowerLevelSectionControl8.disableSection) [not isDisabled1]
  // Moving onto S02 from S01 by train1
  transition from ForwardChoice1 to Stable [position1 = 1] /
    position1 := 12; raise Train2.occupy
  // Moving fully onto S02 from S01 by train1
  transition from ForwardChoice1 to Stable [position1 = 12] /
    position1 := 2; raise Train1.unoccupy
  ...
}
```

Figure 6.9: Important parts of the train model in the MoDeS^3 verification process.

```
statechart Oracle [
  port HigherLevelSectionControl1 : requires SectionControl,
  port LowerLevelSectionControl1 : requires SectionControl,
  ...
  port HigherLevelSectionControl8 : requires SectionControl,
  port LowerLevelSectionControl8 : requires SectionControl,
  port Error : provides Error
] {
  region mainRegion {
    initial Initial
    state Good
    state Error
  }
  transition from Initial to Good
  // Comparing events from the two connected models
  transition from Good to Error when
    !(HigherLevelSectionControl1.enableSection
      == LowerLevelSectionControl1.enableSection) / raise Error.error
  transition from Good to Error when
    !(HigherLevelSectionControl1.disableSection ==
    LowerLevelSectionControl1.disableSection) / raise Error.error
  ...
  transition from Good to Error when
    !(HigherLevelSectionControl8.enableSection
   == LowerLevelSectionControl8.enableSection) / raise Error.error
  transition from Good to Error when
    !(HigherLevelSectionControl8.disableSection ==
    LowerLevelSectionControl8.disableSection) / raise Error.error
}
```

Figure 6.10: The oracle model in the MoDeS^3 verification process.

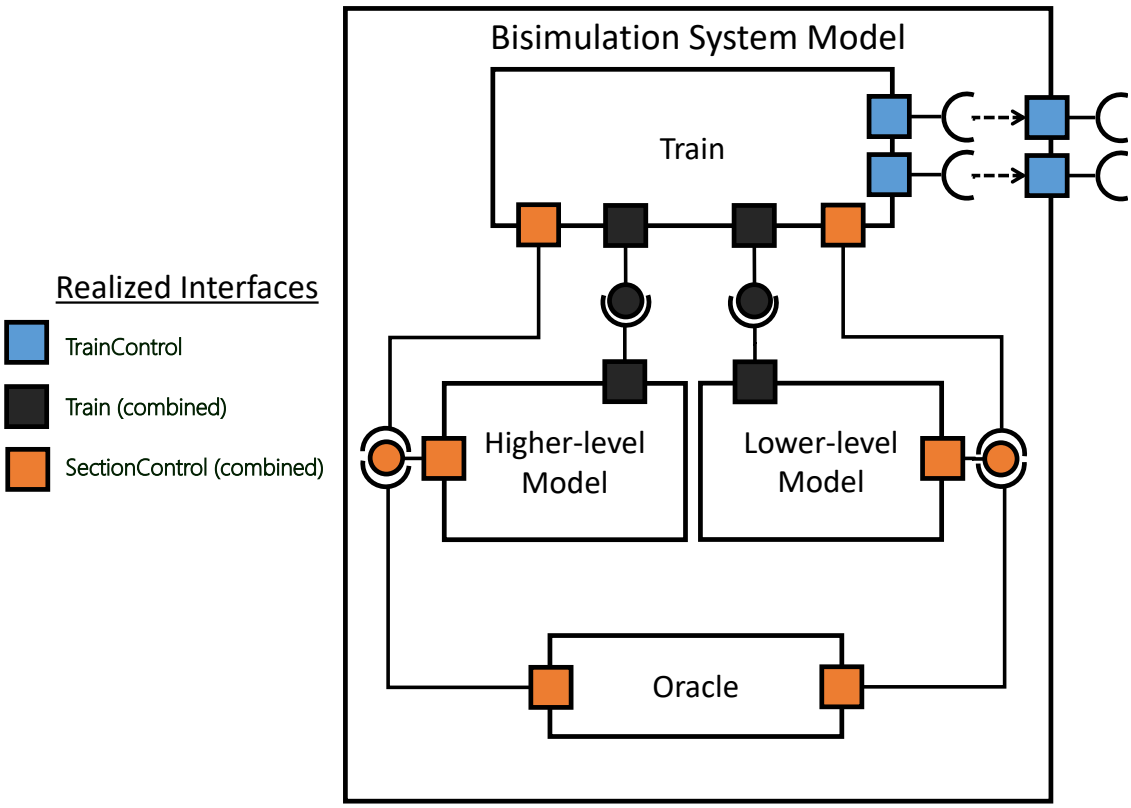

Figure 6.11: A schematic figure about the bisimulation system model. Rectangles represent component instances in the system model. Squares represent ports of component instances, which realize interfaces either in provided mode (lollipop) or required mode (socket). Channels are represented as the connection of lollipops and sockets. Note that ports realizing the *Train* and *SectionControl* interfaces are depicted jointly.

```
cascade BisimulationSystem [
  // Ports for moving the trains
  port Train1Control : requires TrainControl,
  port Train2Control : requires TrainControl
] {
  // The train, track and oracle components
  // The types of components are either HighLevelTrackModel and
  // MediumLevelTrackModel, or MediumLevelZoneModel and LowLevelZoneModel
  component train : Train
  component higherLevelModel : HighLevelTrackModel
  component lowerLevelModel : MediumLevelTrackModel
  component oracle : Oracle
  // Defining the execution order
  execute train, higherLevelModel, lowerLevelModel, oracle
  // Binding train control ports
  bind Train1Control -> train.TrainControl1
  bind Train2Control -> train.TrainControl2
  // Connecting train models to both track models
  channel [ train.Train1 ] -o)-
    [ lowerLevelModel.Train1, higherLevelModel.Train1 ]
  ...
  channel [ train.Train4 ] -o)-
    [ lowerLevelModel.Train4, higherLevelModel.Train4 ]
  // Connecting Train ports from 5 to 8 in case of track models
  ...
  // Connecting the section control ports of the higher level model to
  // both the train model and the oracle
  channel [ higherLevelModel.SectionControl1 ] -o)-
    [ oracle.HigherLevelSectionControl1, train.HigherLevelSectionControl1 ]
  ...
  channel [ higherLevelModel.SectionControl4 ] -o)-
    [ oracle.HigherLevelSectionControl4, train.HigherLevelSectionControl4 ]
  // Connecting SectionControl ports from 5 to 8 in case of track models
  ...
  // Connecting the section control ports of the lower level model to
  // both the train model and the oracle
  channel [ lowerLevelModel.SectionControl1 ] -o)-
    [ oracle.LowerLevelSectionControl1, train.LowerLevelSectionControl1 ]
  ...
  channel [ lowerLevelModel.SectionControl4 ] -o)-
    [ oracle.LowerLevelSectionControl4, train.LowerLevelSectionControl4 ]
  // Connecting SectionControl ports from 5 to 8 in case of track models
  ...
}
```

Figure 6.12: The bisimulation system model in the MoDeS^3 verification process.

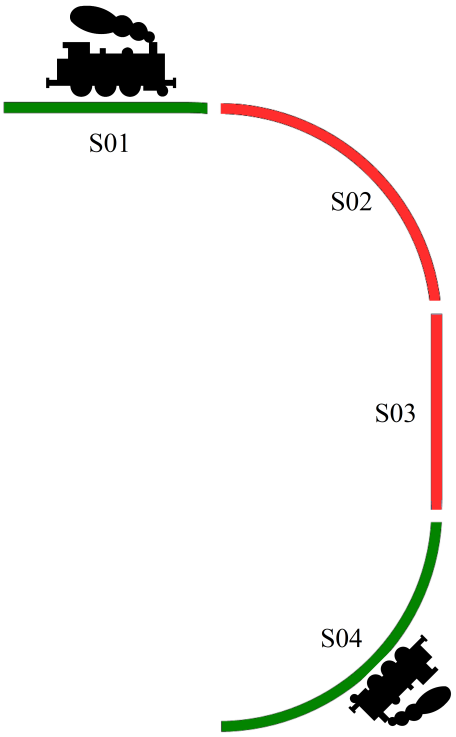

Figure 6.13: The half of the simplified MoDeS^3 track setup used for proving the bisimulation relation between the zone models.

```
cascade HighLevelSystem [
  // Ports for moving the trains
  port Train1Control : requires TrainControl,
  port Train2Control : requires TrainControl
] {
  // The train and track components
  component train : Train
  component highLevelModel : HighLevelModel
  // Defining the execution order
  execute train, highLevelModel
  // Binding train control ports
  bind Train1Control -> train.TrainControl1
  bind Train2Control -> train.TrainControl2
  // Connecting train models to the track model
  channel [ train.Train1 ] -o)-[ highLevelModel.Train1 ]
  ...
  channel [ train.Train8 ] -o)- [ highLevelModel.Train8 ]
  // Connecting the section control ports of the
  // higher level model to the train model
  channel [ highLevelModel.SectionControl1 ] -o)-
    [ train.HigherLevelSectionControl1 ]
  ...
  channel [ highLevelModel.SectionControl8 ] -o)-
    [ train.HigherLevelSectionControl8 ]
}
```

Figure 6.14: The *high-level system model* in the MoDeS^3 verification process.

Chapter 7

Conclusion

The Gamma framework is a modeling tool that supports the hierarchical design, implementation and verification of state-based reactive systems using model-driven software development concepts. Gamma has an extensive language family, 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 and test generation. Furthermore, the extensible architecture of the framework allows additional tools and features to be plugged in.

The main contribution of this work is the design and formalization of the Gamma Composition Language. The GCL builds on the Gamma Statechart Langauge, while extending it 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, tightly-coupled controllers. Cascade composition is practical for designing units with a 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.

The precise semantics of the aforementioned composition modes allowed us to extend the existing code generation and verification functionalities already present in the previous versions of the Gamma framework. Now all elements of the GCL are supported during code generation and most of them during model checking. Furthermore, we have designed the Gamma Test Language, which supports the definition of test scenarios for Gamma components. Regarding test generation, now transition-coverage test-suites can also be generated, in addition to state-coverage test-suites.

As for future work, 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 based on DDS. Moreover, we also plan to extend the framework with additional engineering tools, e.g., MagicDraw, and model checkers, e.g., Theta. In response to lessons

learned in the case study, we intend to provide support for symmetry handling in models, e.g., new element types, such as array and structure elements as well as complex ports that simplify the connection of components with many matching ports. Furthermore, we plan to introduce automation processes to support the presented verification process.

By offering multiple modeling aspects, composition semantics, source code generation and verification features in a single, extensible framework, we hope that Gamma can assist system and software engineers in leveraging the potential of model-driven development. As Gamma is now open-source, we also hope that the results of our research influence and aid fellow researchers and developers in developing their modeling tools.

Acknowledgements

First of all, I would like to express my gratitude to my advisor, Vince Molnár. He has continuously provided me with guidance, valuable ideas and feedback. Without his help I never could have carried out my tasks in this quality.

I am also grateful to the members of the Fault Tolerant Systems Research Group, especially András Vörös, István Majzik, and Zoltán Micskei for their suggestions and continuous support over the past years.

Finally, I would like to thank my friends and family for their support and love.

This work was supported by

- MTA-BME Lendület Cyber-Physical Systems Research Group,
- Új Nemzeti Kiválóság Program 2017-2018, and
- Emberi Erőforrás Fejlesztési Operatív Program (EFOP-3.6.2-16-2017-00013).

Bibliography

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

[2] Diego Latella, István Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Springer, 1999.

[3] Michelle L. Crane and Juergen Dingel. On the semantics of UML state machines: Categorization and comparision. In *In Technical Report 2005-501, School of Computing, Queen's*, 2005.

[4] Michelle L. Crane and Juergen Dingel. UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, pages 97–112, Berlin, Heidelberg, 2005. Springer.

[5] Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, 2009.

[6] Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, and Tiziano Villa. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proceedings of the IEEE*, 103(11):2104–2132, 2015.

[7] Bence Graics. Model-driven design and verification of component-based reactive systems. Students's Association Report, Budapest University of Technology and Economics, 2016.

[8] 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, 2017.

[9] Bence Graics. Model-driven design and verification of component-based reactive systems. Bachelor's thesis, Budapest University of Technology and Economics, 2016.

[10] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma Statechart Composition Framework. In *40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, 2018. ACM, ACM.

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

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

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

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

[15] 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.

[16] 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.

[17] Edmund M. Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[18] Christopher Brooks. Ptolemy II: An open-source platform for experimenting with actor-oriented design, February 2016. Poster presented at the 2016 Berkeley EECS Annual Research Symposium.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] N. Hili, J. Dingel, and A. Beaulieu. Modelling and code generation for real-time embedded systems with UML-RT and Papyrus-RT. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 509–510, May 2017.

[27] Eric James Rapos and Juergen Dingel. Incremental test case generation for UML-RT models using symbolic execution. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 962–963. IEEE, 2012.

[28] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. *OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems*, pages 235–248. Springer, Berlin, Heidelberg, 2014.

[29] Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, and Janos Sztipanovits. Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, pages 471–487, Berlin, Heidelberg, 2013. Springer.

[30] Arne Haber, Markus Look, Antonio Navarro Perez, Pedram Mir Seyed Nazari, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 19–31. IEEE, 2015.

[31] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *CoRR*, abs/1505.00904, 2015.

[32] Ajay Chhokra, Sherif Abdelwahed, Abhishek Dubey, Sandeep Neema, and Gabor Karsai. From system modeling to formal verification. *The 2015 Electronic System Level Synthesis Conference*, 2015.

[33] Preeti Ranjan Panda. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80. ACM, 2001.

[34] Paula Herber. *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata*. Logos Verlag Berlin GmbH, 2010.

[35] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic process algebra: From an algebraic formalism to an architectural description language. In *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 236–260. Springer, 2002.

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

[37] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, Edward A. Lee, and Stavros Tripakis. Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming*, 77(12):1235 – 1271, 2012.

[38] Karolina Zurowska and Juergen Dingel. Language-specific model checking of UML-RT models. *Software & Systems Modeling*, 16(2):393–415, 2017.

[39] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[40] Karolina Zurowska. Language specific analysis of state machine models of reactive systems, 2014.

[41] Ábel Hegedüs, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. *Proceedings of the Software Engineering and Formal Methods (SEFM) 2010*, pages 145–155, 2010.

[42] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2007.

[43] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. 2000.

[44] András Vörös, Márton Búr, István Ráth, Ákos Horváth, Zoltán Micskei, László Balogh, Bálint Hegyi, Benedek Horváth, Zsolt Mázló, and Dániel Varró. MoDeS3: Model-based demonstrator for smart and safe cyber-physical systems. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, pages 460–467, Cham, 2018. Springer.

[45] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A formal semantics for complete UML state machines with communications. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, pages 331–346, Berlin, Heidelberg, 2013. Springer.

[46] F. Wagner. VFSM executable specification. In *CompEuro 1992 Proceedings Computer Systems and Software Engineering*, pages 226–231, May 1992.

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

[48] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017.

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

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

[51] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

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

[53] D. Sangiorgi. *Towards bisimulation*, page 11–27. Cambridge University Press, 2011.

[54] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer, Berlin, Heidelberg, 2008.

[55] S. Balaji and M. Sundararajan Murugaiyan. Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.

Appendix: List of Symbols

Symbol	Description
	Tuples, sets and sequences
$(\ldots)$	Tuple
$\{\ldots\}$	Set
$\lvert\ldots\rvert$	Size of set
$\langle\ldots\rangle$	Sequence
A^*	The set of finite (possible repeating) sequences of the elements of the set A
ε	The empty sequence
$\lvert s\rvert$	The length of sequence s
$s[i]$	The ith element of sequence s
$\sigma(a)$	A permutation of set a
$S_\sigma(a)$	All permutations of set a
	Indices and sizes
i	A generic index variable
j	A secondary generic index variable
a'	Another value for variable a
n	A generic size variable
k	An index used for components
K	The number of components
	Temporal valuations
a'	Next value of a
a^i	The value of variable a in the ith step
	Component types
$\ominus$	Definition of synchronous components ("single-threaded")
$\ominus_k$	The kth synchronous component
Ⓢ	Definition of synchronous composites
Ⓒ	Definition of cascade composites
$\textcircled{?}\rangle\ominus$	Definition of composite $\textcircled{?}$ as a synchronous component
$\oplus$	Definition of asynchronous components ("multi-threaded")
$\oplus_k$	The kth asynchronous component
$\oplus$	Definition of the asynchronous wrapper component
Ⓐ	Definition of asynchronous composites
$\textcircled{?}\rangle\oplus$	Definition of $\textcircled{?}$ as an asynchronous component
a_k	Part a of the kth component (e.g., S_k of $\ominus_k$)
	Events
e	An event
E	The set of events
I	The set of input events
O	The set of output events
$\mathcal{D}$	The domain function
$\mathcal{D}(e)$	The domain of event e
d	An element of a domain
d_i	The ith element of a domain
p	A parameter (of an event)
$\perp$	The empty parameter denoting that an event is missing
$inst(e)$	The instances of event e
$inst_\perp(e)$	The instances of event e including the "null" instance $(e, \perp)$
$\hat{I}$	All input events of constituent components

$\hat{O}$	All output events of constituent components
$\hat{\mathcal{D}}$	All domains of events in constituent components
	Event vectors
v_E	An event vector for events in E
V_E	All possible event vectors for events in E
v_I	An input vector
v_O	An output vector
V_I	All possible input vectors
V_O	All possible output vectors
$v_{\hat{O}}$	The last output of constituent components
$V_{\hat{O}}$	All possible last outputs of constituent components
$\perp_{\hat{O}}$	Empty output vector for all constituent components
$\perp_I$	Empty input vector
	Event sequences
q	A queue (a sequence of event instances)
ω	An output sequence of event instances
Ω	The set of possible output sequences of event instances
	Component parts
S	The set of states
s	A state
s^0	The initial state
T	A state-transition function
t	A single state-transition
	Composite parts
$\mathbf{C}$	Constituent components of a composite
$\rightleftharpoons$	Channels in a composite
$\rightleftharpoons(e)$	The the events linked to event e determined by the channels
X	The order of execution of components of cascade composites
	Async wrapper
e_c	The control event
trig	The trigger predicate
a_s	Part a of the wrapped synchronous component (e.g., S_s of $\ominus_s$)
	Messages
m	A message
e_O	The source event of a message
E_O^{ext}	The output event set of the external component
e_I	A single target event of a message
E_I	Target events of a message
E_I^{ext}	The input event set of the external component
$src(m)$	The source (sending) component of m
ⓔ	The external component representing the environment
	Occurrences
$send(m)$	The occurrence of sending message m
$recv(m, e_I)$	The occurrence of receiving message m on target event e_I
$[t]$	An occurrence of transition t
m_I	A message triggering the occurrence of a transition
m_O	A message generated by an occurrence of a transition
M_O	The messages generated by an occurrence of a transition
$\#occ$	The position of occurrence *occ* in a total ordering of occurrences