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

Designing a Formally Verifiable Action Language for the Modeling of Reactive Embedded Systems

Bachelor's Thesis

Author
Balázs Várady

Advisors
Vince Molnár
Gábor Hicz (evosoft Hungary Kft.)

December 15, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Várady Balázs*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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, 2019. december 15.

Várady Balázs
hallgató

Kivonat

Mindennapi életünk során körülvesznek minket a beágyazott rendszerek. Ezek jelentős része biztonságkritikus rendszer, melyek hibás működése szerencsés esetben komoly anyagi károkhoz vezet, de akár emberéleteket is veszélyeztethet. Az utóbbi időben rohamosan nőtt ezen rendszerek komplexitása, ennek ellensúlyozására egyre inkább teret hódítanak a modell-alapú fejlesztési paradigmák. Ezek számos előnye között szerepel, hogy a rendszermodellekből automatikusan származtathatóak bizonyos dokumentumok, mint dokumentáció, implementáció vagy egyéb, formális verifikációhoz alkalmazható modellek. Ez azonban azt feltételezi, hogy a modellek rendelkeznek pontosan definiált szemantikával, ami rendszerint nem igaz a mérnöki munka során alkalmazott modellekre – sokszor szándékosan, néha viszont nem.

A helyes működés bizonyításának lehetősége alapvető fontosságú követelmény biztonságkritikus rendszerek tervezésekor. A tesztelésen kívül formális módszerek is alkalmazhatóak a viselkedés helyességének kimerítő és automatizált elemzéséhez, rendszerint már a rendszer fejlesztési folyamatának korai szakaszában is. Ennek egy gyakran alkalmazott módja a modellellenőrzés, melynek során az alkalmazott ellenőrző eszköz a rendszer állapotterét kimerítően elemzi. Ez kézenfekvő módja az állapot alapú modellek vizsgálatának is, melyeket gyakran alkalmazzák reaktív rendszerek belső viselkedésének modellezésére.

A Gamma Állapotgép Kompozíciós Keretrendszer egy szoftverfejlesztést segítő keretrendszer, mely támogatja a modellvezérelt szoftverfejlesztés paradigmáit. Az elkészített modellek vizsgálatához rejtett formális módszereket alkalmaz azáltal, hogy a támogatott magas szintű modellező elemek szemantikáját precízen és formálisan verifikálható módon definiálja. Jelenleg a külvilág eseményeire adható reakciókat leíró nyelve erősen korlátozott.

Jelen dolgozat célja egy formálisan verifikálható akciónyelv tervezése állapot alapú modellek számára, majd ennek integrálása a Gamma Keretrendszerbe. A nyelvnek képesnek kell lennie a rendszer reakcióinak részletezésére, mégpedig kellően nagy kifejezőerővel ahhoz, hogy le tudja írni a beágyazott rendszerektől elvárható viselkedést, ugyanakkor formális verifikációra alkalmasnak is kell maradnia. Ez azt jelenti, hogy a nyelvnek a véges állapotgép formalizmus határain belül kell maradnia, miközben támogatnia kell magas szintű nyelvi elemeket a kifejezőképesség növeléséhez. Ezt úgy érjük el, hogy megvizsgálunk különböző eszközöket, melyek alkalmasak reaktív beágyazott rendszerek modellezésére és így szerzett tapasztalataink alapján olyan nyelvi elemeket definiálunk, melyek nem lehetetlenítik el a formális verifikációt.

A másik szempont, amit figyelembe kellett vennünk a nyelv tervezése során a precízen definiált szemantika szükségessége a definiált nyelvi elemek számára. Ezt denotációs szemantika megadásával érjük el, vagyis definiáljuk a transzformációt egy másik formalizmusra. Ezen transzformációk célja a jelenleg kísérleti xSTS formalizmus, amelyet a Theta Keretrendszer fog tudni verifikálni. Ezen kívül Java kód is generálható belőle egyszerűen, amint azt látni fogjuk.

A nyelv széleskörű funkcionalitását, alkalmazhatóságát és korlátait egy esettanulmány során vizsgáljuk, melyben egy RPN számológépet modellezünk.

Abstract

In our everyday lives, we are surrounded with embedded systems. A siginificant proportion of these systems is safety-critical, such as cars, trains, airplanes, etc. The faulty behavior of these systems could result in at least serious financial losses, if not threaten human lives. As these systems are getting more and more complex, the application of the *model-driven paradigms* is gaining more and more ground in their development processes. Among its numerous advantages, model-driven software development enables the generation of various artifacts based on the system model, such as documentation, implementation and different models suitable for verification. This however, assumes precisely defined semantics, which is often missing in models used in engineering, at times intentionally, at times not.

The feasibility of proving the correctness of a model is an important requirement when designing safety-critical systems. In addition to testing, *formal methods* can be applied to verify the correctness of behavior exhaustively and automatically, even in the early phases of designing systems. A common means of formal verification is *model checking*, during which the state-space of the given system is exhaustively analyzed. It is also convenient for the analysis of state-based models, which are commonly used in engineering to model the behavior of reactive systems.

The Gamma Statechart Composition Framework is a software development framework supporting the model-driven software development paradigm. It applies hidden formal methods to the created models by supporting modeling on higher abstraction levels with precisely defined, formally verifiable semantics. However, its language for describing actions in reaction to various events of the outside world is strongly limited.

The goal of this work is to design a formally verifiable action language for state-based models and integrate it into the Gamma Framework. The language should be able to detail the reactions of the system with a great enough expressive power to describe the behavior common to embedded systems, but it should also be formally verifiable. This means, that the language must stay within the boundaries of finite-state machines, but also support high-level language elements to increase its expressive power. This is achieved by analyzing tools applicable for the modeling of reactive embedded systems and defining language elements based on constructs observed in them that do not inhibit formal verifiability.

Another aspect that had to be taken into account is the precise definition of the semantics of the language elements. This is done by providing denotational semantics for each one of them by defining model transformations to the experimental xSTS formalism used by the Theta Framework, which can be used for formal verification and also Java code generation, as demonstrated in this work.

The extensive functionality, applicability and boundaries of the language are demonstrated through a case study of an RPN calculator.

Chapter 1

Introduction

Due to the increasing complexity of embedded systems, and software systems in general, the design, implementation and analysis of these systems is getting more and more difficult directly handling the source code. Thus, the modeling of these systems should happen on a higher level of abstraction.

In theory, it would be possible to automatically generate source code and formally verify the correctness of behavior solely using the models of a given system. This is the goal of the *model-driven software development* methodology. However, engineering models made for a certain purpose (e.g. communication, visualization or documentation) often lack the precisely defined semantics, or at times even syntax required for these purposes. In some cases, like the UML 2.1.2 standard, *"implementors may provide [...] informal feature support statements [...] for less precisely defined dimensions such as presentation options and semantic variation points"* [10], which feature deliberately discards the mathematical strictness in exchange for ease of communication. Attempts have been made to tackle this problem and define variability within modeling languages, for instance in [6].

Some modeling languages, especially those used in formal methods, possess not only the syntax and metamodel, but also the semantics and well-formedness constraints in a strict mathematical manner. Examples for these models include most mathematical models of computation (Turing machine, finite-state machine) or the extended timed automata of the UPPAAL tool. The drawback of these models is the difficult application for engineering tasks, as they operate on lower abstraction levels, often with limited expressive power and strongly mathematical syntax.

Reactive systems can be described using several kinds of *behavioral models*. One of the most practical ways of modeling these systems is the *statechart* formalism – one formalism for state-based models. The reason for this practicality is the relative simplicity and visual representation of the models, that facilitates not only the design, but also the analysis of these systems. The Gamma Statechart Composition Framework, deatiled in [18] also applies this formalism for modeling component-based reactive systems, with an additional goal of supporting hidden formal verification techniques on the system models.

The specification of the reactions of these systems to the events of the outside world is most efficient in a procedural manner. For this reason, most statecharts have some sort of *action language* to bridge the gap between state-based and process-based modeling. The different action languages of existing modeling tools are usually designed to solve a particular problem: some can be directly used for execution or code generation, some can be used for formal verification, and some are useful for communication at the expense of precisely defined semantics.

This thesis will introduce an action language to the Gamma Framework, with the intention to address this situation and extend the above mentioned tool with a both mathematically precise and from an engineering standpoint convenient feature. Formal verification and execution is achieved through the definition of precise *denotational semantics* – i.e. by transforming the model to various models designed for specific purposes.

An important aspect taken into account when designing the action language was the ease of use and the convenient description of the behavior common to embedded systems, by supporting well-known control statements and data structures of 3rd generation programming languages. Examples for these control statements are the *if-else*, *switch-case* and *for* statements. For data structures, currently *array* (indexable groups of variables of the same type) and *record* types (groups of variables of possibly different types) are supported in addition to primitive types. This results in a C-like programming language, with some features resembling the UML textual syntax and a fairly high expressive power. Turing completeness is intentionally avoided to ensure the decidability of the verification problems. The language is efficiently verifiable using formal methods, as – by not being computationally complete – the algorithms written in this action language are always guaranteed to terminate. Thus, it is possible for a model checker to exhaustively analyze the state-space of the system and prove or disprove the correctness of its functionality.

It is integrated into the Gamma Statechart Composition Framework, using its type system and expression language, also being extended by it with statechart-specific elements, such as *event raising* and *timeout*-related statements. Automated transformations are currently supported to Java and xSTS codes – the latter being used by the Theta Framework detailed in [19] –, enabling the execution and formal verification of the designed system models.

The rest of the thesis is structured as follows. Chapter 2 presents the theoretical background behind modeling formally verifiable reactive systems. It also explores the existing modeling tools, with special regard to their action languages, and also the targeted framework used by and using the action language detailed in this work. Chapter 3 describes the various features and capabilities of the action language, describing the syntax, semantics and the motivation behind each of its constructs, as well as providing the reader with examples of the usage of each of the language elements. Chapter 4 presents the tools and details of the implementation and integration into the Gamma Framework. The applicability and usefulness of the introduced action language is discussed in Chapter 5, along with the validation methods applied to each of the contained elements. Finally, Chapter 6 provides concluding remarks and possibilities for further improvement.

Chapter 2

Background

This chapter introduces the concepts necessary to understand the requirements and ideas behind the rest of the work. In Section 2.1, we take a look at the applicability of modeling in the design of safety-critical reactive systems, followed by the corresponding formal verification techniques in Section 2.2. Then, in Section 2.3, a short introduction is given on behavioral modeling, leading to the precise definition of the statechart formalism in Section 2.4, which is commonly used to represent behavioral models and which is a core element of the Gamma Framework. After that, in Section 2.5, we examine the action languages of existing tools with respect to their expressive power and possibilities of formal verification. Lastly, in Section 2.6 a brief introduction of the Gamma and Theta frameworks is given, which are the target environments of the action language described in the following chapters.

2.1 Model-driven software development

Due to the application of the modeling concept in several completely different domains, first of all, we need to define the meaning of *model* in this work.

Definition 1 (Model). A model is the simplified image of an element of the real or a hypothetical world (the system), that replaces the the system in certain considerations. ▪

For a model to be interpretable, executable or formally verifiable, it must be described according to predefined rules in the given domain. This set of rules is provided by *modeling languages*.

Definition 2 (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.
- *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 either *operational* (what should happen during execution) or *denotational* (given by translating concepts in a modeling language to another modeling language with well-defined semantics). ▪

Models can grasp various aspects of a system. Structural models describe the structure of the system, representing knowledge regarding the parts of the system and the properties and connections of these parts. This means that the model describes static knowledge and not temporal change. On the other hand, behavioral models describe the change of the system over time through its changing of states and execution of processes. These categories do not cover every aspect of a system, and usually cannot be separated this well in practical applications. For instance, action languages of state-based models describe the behavior of the system in a procedural way. There are several possible formalisms for both kinds of models, some of which are discussed in Section 2.3.

Model-driven software development (MDSD) is a software development methodology that utilizes domain models as the primary artifact throughout the entire software development process. This approach splits the traditional development process into two parts. The first part is infrastructure development, during which the modeling language, platform and transformations are defined. The second part, application development, involves only modeling in the application domain. This strongly simplifies the design phase by efficient reuse of code and early validation. The major advantage of this methodology is the productivity improvement during the development of a (software) system, achieved through the fact that it is possible to derive different design artifacts, like documentation, source code, configuration and even other models from the system model [21]. One possible implementation of MDSD is the so-called Y-Model devleopment process (see Figure 2.1). The Gamma Framework also supports this methodology.

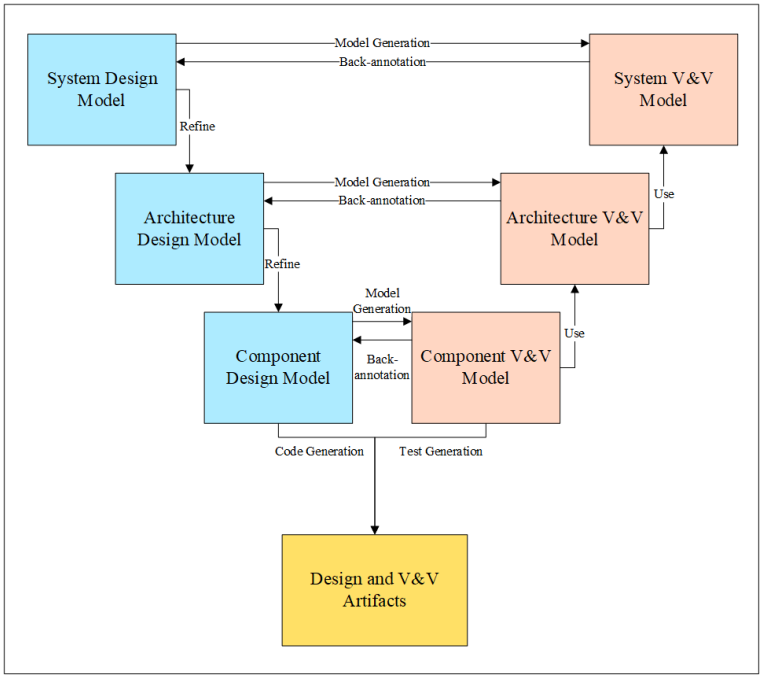


Figure 2.1: The schematic description of the Y-Model

The process of deriving design artifacts is called *model transformation*.

Definition 3 (Model Transformation). Model transformation is the process of generating the target model from the source model. This process is described by by a transformation definition consisting of transformation rules, and a transformation tool that executes them. A transformation rule is the mapping of elements of the source model to the elements of the target model. [15] ▪

Model transformations can be categorized based on the types of the source and target models: model-to-model (M2M), model-to-text (M2T), text-to-model (T2M) and text-to-text (T2T). These categories fundamentally define the tools required and usable for handling the different models.

There are also two important factors to consider when designing a model transformation:

- *Consistency*: the same structure or behavior is described by the source and the target models (in their respective domains).
- *Traceability*: the images of the original elements of the source model can be traced back to the original elements, from which they were generated.

2.2 Formal verification of systems

Formal verification is the act of comparing the design model of a given system against certain requirements formulated by the stakeholders. This process requires mathematically precise design models and requirement specifications, but guarantees mathematical precision in its result as a proof of correctness or counterexample of the correct behavior.

Model checking [7] is an automatic formal verification technique for a finite-state model of a system, which explores the state space of the given model soundly and often exhaustively. This results in a complete analysis of the behavior of the given system unlike in case of simulation or testing, which can only sample it. The *model checker* takes the formal model of the system and the formal requirement (a formula usually given in some kind of mathematical logic, which has to be satisfied by the system) as an input, and returns the result of the evaluation often with a witness proving the result (see Figure 2.2).

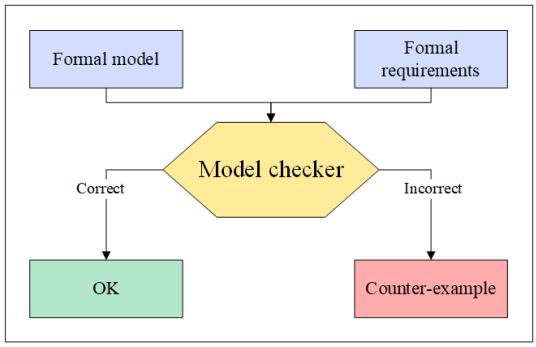


Figure 2.2: The schematic description of the model checker

As the formal verification of the correct behavior is often a requirement against critical systems, the ability to transform the design model of the system into other models - in this case a verification model - proves to be a particularly useful quality. The precise formal analysis of these models either confirms the correctness of the given model, or provides example traces where the model does not meet the requirements. These faults can then

be back-annotated to the high-level models, i.e. the tool can automatically identify the faulty section of the original model. This enables the system designer to make corrections, even in the early phases of the development process [21].

2.3 Formalisms for modeling behavior

Different kinds of behavioral models grasp different aspects of the modeled system. *State-based models* focus on the current state of the system and the change of states in response to the events of its environment. The process of this change is secondary, and these models consider these so-called transitions atomic and instantaneous. Examples of this kind of modeling include UML State Machines [16] or statecharts [14]. On the other hand, *process models* focus on the series of actions a system takes in order to achieve its goal, and these actions possess a temporal extent. Examples of this kind of modeling include UML Activity Diagrams [16], but the control flows of programs implemented in the popular imperative programming languages (e.g. C/C++, Java) are usually also process models. As these modeling languages often do not have precise semantics, it is common practice to define it by means of model transformations to formal models (denotational semantics). Several general-purpose imperative programming languages and process models used for their visualization can be easily transformed into Turing machines and many state-based models – such as statecharts – correspond well to finite-state machines.

One of the most important questions in the verification of computer programs is the ability to decide the termination of the given program – known as the *halting problem*. This is necessary, as to prove its correctness, the model checker analyzes the *reachability* of certain states, for which it must traverse each possible execution of the given program. For Turing-complete models, the reachability of a given state is equivalent with the halting problem, and it is proven in [20] that in general (i.e. for Turing machines), it is not possible to decide the termination for all program-input pairs. However, for finite-state machines, the problem is solvable [17]. For this reason, it might be beneficial to restrict the system model to a finite-state machine for the sake of formal verifiability, especially in case of safety-critical systems, even if it possesses less expressive power than other types of automata.

According to automata theory, finite-state machines can be considered a specialized form of pushdown automata, which, in turn, are a specialized kind of Turing machines (see Figure 2.3). This, through the associations made earlier in this section, provides us with an excellent basis of comparison for different types of system models.

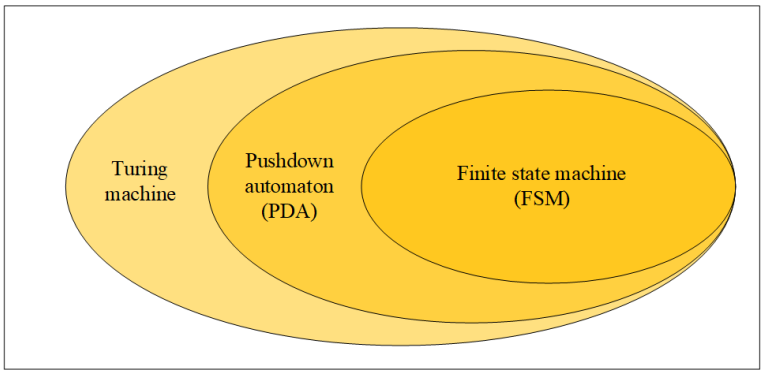


Figure 2.3: Classes of automata

2.4 Statechart formalism

Although in engineering the *statechart* formalism is the more convenient way to describe reactive systems, the mathematical basis of this modeling language is the so-called *finite-state machine*.

Definition 4 (Finite-state Machine). A finite-state machine (FSM) is a model of computation to describe the behavior of a system in reaction to the events of its environment [2]. Mathematically, a finite-state machine is $M = (S, s_0, I, O, T)$ where:

- $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is a finite set of states,
- s_0 is the initial state,
- $I = \{i_0, i_1, \ldots, i_p\}$ is a finite set of input events, that are stimuli from the environment,
- $O = \{o_0, o_1, \ldots, o_q\}$ is a finite set of output events, that are stimuli for the environment,
- $T : (I \times S) \to (S \times O)$ is the transition function, that represent changes of states in response to input events and generating output events.

A possible sequence of steps can be described through a *trace*. A trace is a finite sequence of alternating states and transitions beginning and ending in a state: $\varrho = s_0, t_0, s_1, \ldots, t_{n-1}, s_n$, where $s_i \in S$ and $t_i \in T$. ▪

The ***statechart*** [14] formalism is a possible extension of this mathematical model with the following elements. Conforming to the UML Standard [16], it includes the following extensions to finite-state machines:

- States
  - hierarchical states: states that themselves have a state space.
  - concurrent regions: a state is active in each of the regions, which operate asynchronously.
  - actions assigned to states (entry, exit, do): behaviors that happen when a state becomes, is, or stops being active.
- Pseudo-states
  - initial state: a state which is only active when the execution of the region starts.
  - final state: a state which is only active when the execution of the region ends.
  - history states (deep and shallow history): states that are active when the execution of a region starts, with the next state being where the execution of the region previously stopped.
- Transitions
  - given in the form: trigger [guard] / action, that determine the possible following states depending on the state of the system and the incoming events, also describing the reaction of the system during the change of states.

– complex transitions (fork-join, branching): for handling concurrency and decisions.

The syntax of the possible actions is detailed in an *action language*, which is different from tool to tool, along with some further extensions. The features of this action language determine whether the system model corresponds to an FSM or a more general type of automaton – with the corresponding expressive and verification capabilities.

The action language designed in the following chapters has the goals of having a fairly high expressive power and supporting formal verification. According to the previous section, the compromise is to remain at the complexity of FSMs by only supporting high-level imperative programming constructs that possess at most the expressive power of that formalism.

2.5 Related Work

To be able to determine the best-practices and dead-ends, we have to analyze the action languages of existing tools. We are going to use the definition of action provided in the UML Standard [16].

Definition 5 (Action). An action is a fundamental unit of computational behavior specification. An action takes a set of inputs and produces a set of outputs. It may also modify the state of the system.

It is important to note that the sets of inputs and outputs may also be empty. ▪

In the following sections, we are going to inspect various modeling tools, with special attention to their action languages, or, if not explicitly defined, their process-oriented reaction description capabilities.

2.5.1 UML 2

The Unified Modeling Language (UML) [16] is a general-purpose graphical modeling language for the domain of software engineering. In recent years, it has become the standard way to visualize the design of software systems. Examples of UML diagrams include use-case diagrams, class diagrams, sequence diagrams, activity diagrams, state machines, etc.

According to the UML Standard [16], the means of describing actions is through Actions (with capital A). Actions are defined as in Definition 5. Actions (ActivityNodes) are always contained in Behaviors (Activities or Interactions), that define the context of the execution of the Action. The execution of an action represents some transformation or processing in the modeled system. The standard defines several types of actions (specializations of ActivityNode), for instance Invocation Actions, Object Actions, Link Actions, Structural Fearute Actions, Variable Actions, Accept Event Actions. One special kind of Actions are Structured Actions (StructuredActivityNodes), which are not only Actions, but also ActivityGroups. This means, that Structured Actions can contain Activities (a graph of ActivityNodes and ActivityEdges defining the control and data flow of a Behavior) and Variables. Besides that, another construct called Expansion Region is available, which has an input collection (a given set of input data for the region), and executes the contained StructuredActivityNode for each of the elements of the input collection, each time providing the current element as input data.

In addition to the Actions defined in the standard, OpaqueActions are also available for modeling actions. OpaqueActions are Actions, for which the specification may be given in a textual syntax other than UML. It consists of body and language strings. If multiple of those elements are available, the standard does not determine how the choice is made. This is one example of syntactic and semantic variation points and results in the modeling of arbitrary actions, making the language extremely versatile.

Variability of syntax and semantics is deliberately included in the modeling language, as discussed in [6]. This promotes the communication of engineers and the visualization of systems, but results in an incomplete definition of the language. Therefore, UML models in general are not suitable for code generation or formal verification.

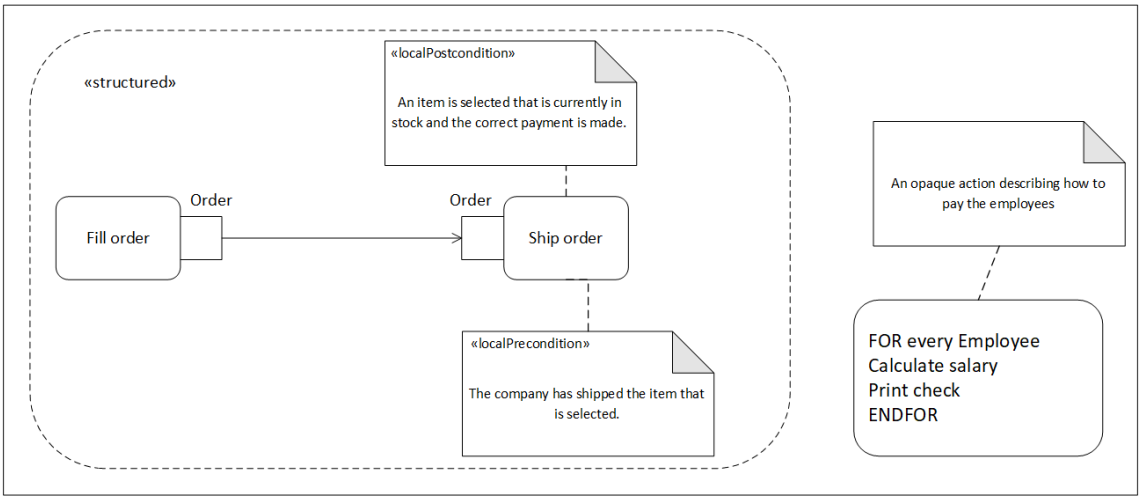


Figure 2.4: A Structured Activity (left) and an Opaque Action (right) according to UML 2.5.1

2.5.2 fUML and Alf

Foundational UML (fUML) [13] is a computationally complete subset of the UML 2 standard with a precise definition of the operational semantics of that subset and serving as a foundation for higher-level UML modeling concepts. It defines a basic and general virtual machine for UML, enabling the transformation of the models into executable forms for verification, integration and deployment. It attempts to maintain a degree of genericity in order to support various execution paradigms, environments and underlying data structures. This is achieved by leaving certain key semantic elements (e.g. time, concurrency, inter-object communication) unconstrained and explicitly defining semantic variation points (e.g. event dispatch scheduling, polymorphic operation dispatching). The majority of the standard is defined by excluding elements of the UML2 standard.

fUML offers similar mechanisms for describing actions as UML, applying several restrictions to the elements of the latter. For instance, custom constraints are not supported in fUML, therefore, local pre- and postconditions are not supported either. Redundant elements are also excluded from the standard, like ActionInputPins and ValuePins, as they can be represented using object flow. Variable Actions are also excluded, as Variables are not supported in fUML – they too can be represented using object flow. Structured Actions are still supported with the exclusion of Variables and Sequence Nodes (that would execute a sequence of ExecutableNodes in order), similarly to Expansion Regions with

no exclusions. However, Opaque Actions are excluded, as, being opaque, they cannot be executed.

An alternative to this is the Action Language for Foundational UML (Alf) [12]. It is a textual surface representation for UML modeling elements, mapping a textual concrete syntax to the abstract syntax of the fUML standard. The main focus of Alf is to be an action language, but it also provides concrete syntax for modeling structural elements of fUML. It possesses a naming system based on UML, an implicit type system for activities and also the expressivity of OCL [11] with a C-like syntax, also allowing the UML textual syntax whenever it exists (e.g. type declaration of variables after the name and a colon, as opposed to before the name of the variable).

To examine the formal verification capabilities residing in fUML and/or Alf, it suffices to analyze one or the other, as Alf is directly mapped to the fUML abstract syntax. In comparison to UML2, the majority of semantic variation points are either precisely defined or eliminated, thus, models created according to these standards are transformable into executable ones, enabling verification. Nonetheless, due to the Turing completeness of fUML, the termination of the possible algorithms cannot be guaranteed - the language is *undecidable*. Hence, the correctness of the models with regard to the requirements cannot be formally verified in all cases by a model checker.

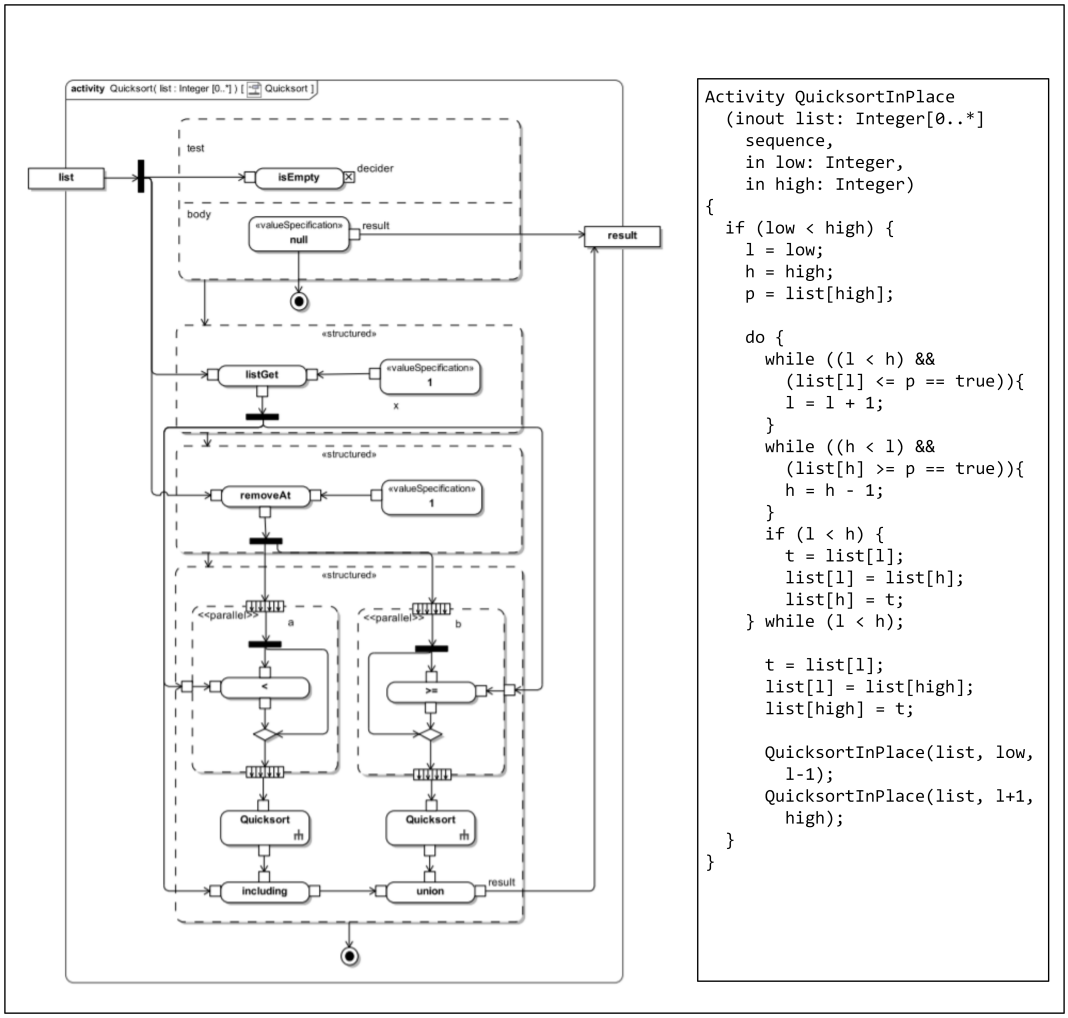


Figure 2.5: Quicksort algorithm using fUML (left) and Alf (right) from Appendix B of [12]

2.5.3 Yakindu Statechart Tools

YAKINDU Statechart Tools is a modular toolkit for the development and analysis of reactive, event-driven embedded systems using the statechart formalism, already discussed in Section 2.4. It combines graphical and textual syntax for the modeling of systems, and, among others, supports the simulation and source code generation for the designed statecharts. Yakindu consists of the following features for dealing with statecharts: statechart diagram editor, statechart simulator, code generators, generator projects for custom model transformations, validator, unit testing framework, C language integration and a debugger.

In Yakindu statecharts, transitions may have reactions in the form *trigger [guard] / effect*. Also, states may possess entry effects and exit effects in the form *entry / effect* and *exit / effect* respectively, and local reactions containing effects, such as *every 1s [guard] / effect*. The elements permitted in the specification of effects are called actions, and together, they constitute the action language of Yakindu. An effect contains zero, one or more actions, separated with semicolons. Normally actions are statements, which are either assignments, event raisings, or operation calls. Assignments in Yakindu assign a given value to a given variable, possibly evaluating an arithmetic or logical expression meanwhile. Event raising takes place using the *raise* keyword and the name of the event to be raised. Operations are functions declared in a C header (.h) file and implemented outside of Yakindu, which is then available to be called from a statechart using an operation call. An operation is called similarly to other programming languages, using the name of the operation and passing the actual arguments in parentheses.

Regarding the possibilities of the verification of Yakindu statecharts, the editor offers some verification capabilities in the form of a simulator and a unit testing framework. However, these tools are able to analyze the behavior of a certain system model through manual sampling only. Formal verification capabilities are currently not available in the tool, although it is possible by transforming Yakindu models to models verifiable by other tools, such as UPPAAL – similarly to the workflow of the Gamma Framework. Naturally, operation calls can only be formally verified if the target language can be formally verified, which is in general not possible for the C language.

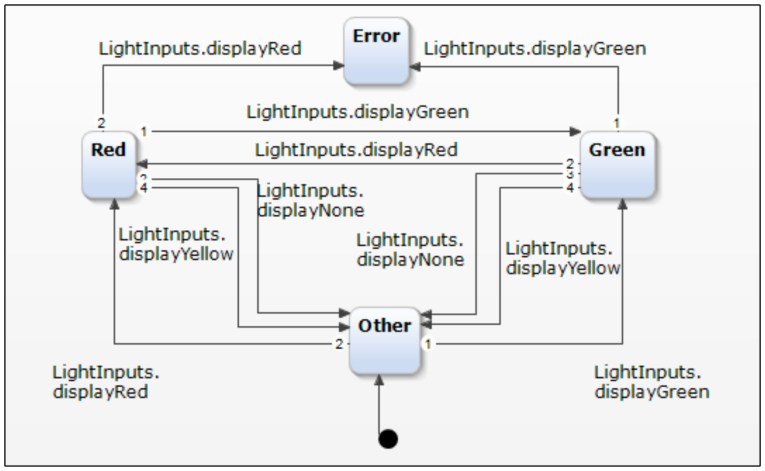


Figure 2.6: Yakindu statechart containing *raise event* actions

2.5.4 UPPAAL

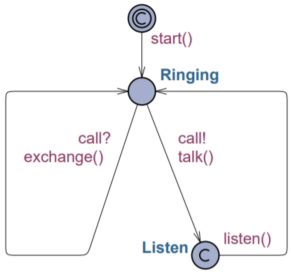
UPPAAL is a tool for modeling, simulation and verification of real-time systems [8, 3]. UPPAAL uses the timed automata formalism described in [4], extended with integer variables, structured data types, user defined functions and channel synchronisation. The main focus of the tool is the model-checking of systems modeled as networks of timed automata.

For the complete description of the capabilities of the UPPAAL tool, we refer the reader to [8], here only follows a short summary of the referred paper for clarity of the following paragrahs. In UPPAAL, the root element is the Network of Timed Automata, which contains Templates. Templates are models of (extended) timed automata, which can be instantiated, then called Processes. Templates have parameters that can be bound to values during instantiation, locations (possible "situations" of the automaton), and edges (conforming to the semantics of transitions in standard timed automata). UPPAAL also allows the use of variables and clocks, and the 3-tuple (location, variables, clocks) defines the state of a Process. Asynchronous automata can interact by using synchronization channels. Locations may contain invariants – Boolean-evaluated expressions that must always hold when the automaton is in the given location. Edges may contain channel synchronizations, guards, updates and selection expressions. From UPPAAL 4.0, user-defined functions – defined using a specialized subset of the C language – are also allowed.

As we can see, there are no explicitly defined actions in UPPAAL, therefore, we consider the process of the reaction of the system an action, based on Definition 5. This is met by updates and selection expressions found on edges. Updates are expressions with side effects that change the state of the system, such as assignments. Selection expressions non-deterministically bind a given identifier to a value in a given range that is valid in the scope of the given edge – basically being a special kind of local variables. It is important to note, that user-defined functions may appear as part of these expressions. For functions without side effects, the evaluation is equivalent to the evaluation of expressions. For functions with side effects, however, the evaluation of the effect could become a more complex task.

UPPAAL is a tool for the formal verification of the above mentioned systems, so the ability of verification of the modeled systems follows naturally from the nature of the tool. The evaluation of edges is atomic, which is extremely important in case of user-defined functions: the syntax permits many constructs from C (except e.g. pointers) extended with timed automata-specific elements. However, these elements are defined in a way that ensures the termination of these functions – e.g. allowing only foreach-loops, or bounded integers. Naturally, this might have an effect on the performance, but does not prevent verifiability in general.

An example of an UPPAAL automaton can be seen on Figure 2.7, based on an example in [8]. It aims to model the behavior of 'girls' in the following example: let there be n girls, each aiming to share their private secrets with all the others. Any girl can call anyone, and after a conversation, both know all the secrets of both of them. The question is, how many calls are necessary in total, so that in the end each of the girls knows everything. Based on this description, the girls in the model can demontrate two different behaviors: either call another girl (represented by sending a channel synchronization), put their secrets in a common temp variable, then copy the 'merged' secrets when the other one also put their secrets in, or wait for a call (by waiting for a channel synchronization), then merge their secrets with those of the other one, and copy the secrets for themselves.




```
int secrets;

void start() { secrets = 1 << id; }

void talk() { tmp = secrets; }

void exchange() { secrets = (tmp |= secrets); }

void listen() { secrets = tmp; }
```

Figure 2.7: Template automaton of the Gossiping Girls problem along with the corresponding function definitions

One important thing to note about the timed automaton formalism of UPPAAL is the fact that it results in models that are difficult to implement, its constructs being abstractions of real-world behaviors – e.g. channel syncronizations or the bitshift and bitwise OR operators in the example. Therefore, it is impractical to model complex systems in UPPAAL. A viable alternative is transforming existing system models to UPPAAL timed automata for verification, which is exactly what the Gamma Statechart Composition Framework does.

2.6 Target Technologies

2.6.1 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework [18] is a tool that facilitates the design, verification, validation and code generation for component-based reactive systems. It defines its own, domain specific statechart composition language for assembling the system model from its components modeled using the statechart formalism. It aims to fill the gap between UML tools, which are used for visualization of the composition of systems, and the various statechart modeling tools, such as Yakindu, that enable system designers the modeling of reactive system components.

The Gamma Framework offers the Gamma Composition language (GCL) for the modeling of components, interfaces, ports and communication channels. Components can be either composite components – which define a composition of components by instantiating other components, binding their ports to the ports of the composite component and defining communication channels between components – or statecharts, which are the basic building blocks defined through the Gamma Statechart Language (GSL).

In GCL, the components communicate through ports, through which they send or receive certain predefined signals, called events. Events are declared on interfaces, which are realized by ports. They can also be parametrized, passing additional data along with the basic signal. This functionality is implemented using variables – boolean-type for the events, and the appropriate type(s) for the parameter(s) –, that can be accessed on both ends of the channel. Channels are always one-directional, the sending and the receiving end are specified at the interface realizations. There are also special events, called timeout events, which fire when a previously set timeout variable signals at the end of the specified time interval.

The GSL can be used to define statecharts of the Gamma Framework. This formalism provides a textual representation for the modeling of statecharts, unlike the ordinary graphical representation. Gamma Statecharts can be parametrized, and can also have ports – interfaces which they realize as required or provided. They may have variables (which are visible in the scope of the statechart), timeout variables, transitions and a region. The region contains the states, between which the transitions can run. Transitions may have triggers (the events to which the transitions fire), guards (additional conditions for the possibility of the firing of a transition) and several actions (reaction specifications of the system). States always have names and may contain invariants, entry and exit actions, and also additional regions.

The framework also supports executable code generation for the modeled systems. Gamma can generate source code not only for the modeled statecharts, but also for the composition code on top of these components, due to the precisely defined compositional semantics. Also, it allows the integration of external code generators. The currently supported target language is Java.

The numerous features of Gamma also include the application of hidden formal methods for the verification of the modeled systems. It is especially useful combined with the high-level user interface with elements designed for efficient model checking. For comparison, the modeling of systems in other tools also offering formal verification is often difficult, due to the low-level modeling elements these tools offer.

Gamma also allows the integration of third-party statechart modeling tools and model-checkers, currently containing an implementation for the integration of Yakindu Statechart Tools discussed in Section 2.5.3, UPPAAL discussed in Section 2.5.4, and the transformation for the Theta verification framework (Section 2.6.2) is under development. Back-annotation of the generated models is also supported for these tools.

While there are tools with similar goals, the distinctive quality of Gamma is supporting all the above mentioned features by defining an intermediate model that allows the integration of third-party tools with different target functionalities in the domain of critical systems.

An example of a Gamma statechart can be seen on Figure 2.8.

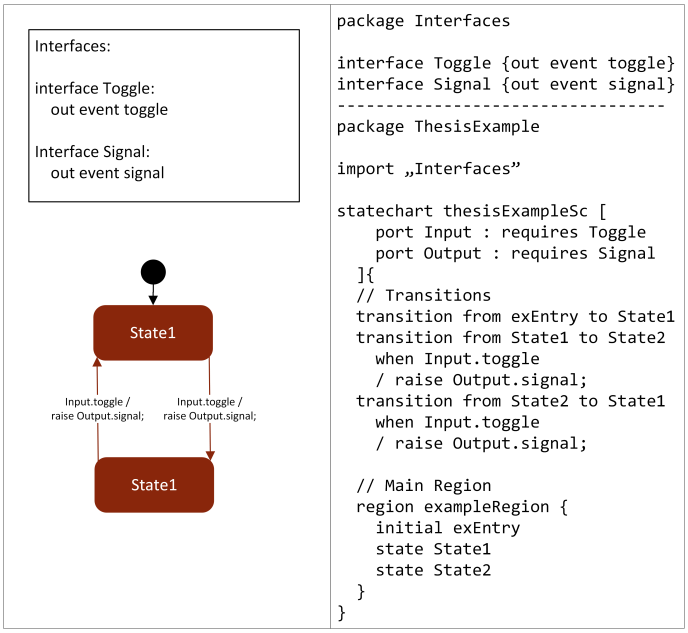


Figure 2.8: A Gamma statechart (right) and the corresponding visual representation (left)

2.6.2 Theta

The Theta Framework [19] is a model checking framework that supports the development and evaluation of abstraction refinement-based algorithms, which are, in turn, useful for the reachability analysis of different formalisms. It aims to solve the problem of having to use different model checking tools for different analysis algorithm and modeling formalism pairings, by offering a generic solution, which is freely extensible. The rationale behind this solution is the fact that no single configuration of formalisms and analysis algorithms can verify all models, and the execution times also vary greatly.

The framework itself is generic, modular and configurable, which allows researchers to implement, evaluate and compare new components and combinations. The genericity and modularity allow for new components to be freely implemented, and the configurability results in the possibility to use and evaluate any of the implemented combinations.

The main parts of the framework are the formalism and language-front ends, the analysis back-end and the SMT solver interface. The first two can be extended and configured by the user, and the latter can be used by the analysis components for communication. In addition, Theta contains three concrete, built-in tools for the verification of transition systems, control flow automata, and timed automata.

Chapter 3

Theoretical Results

In this chapter, the various aspects of the proposed action language are detailed. In Section 3.1 the different elements and their intended semantics along with their syntaxes, in Section 3.2 the well-formedness constraints in the form of validation rules. In Sections 3.3, 3.4 and 3.5 the denotational semantics are presented by detailing setps of the transformation of the abstract syntax tree to various other models, namely to the xSTS formalism of the Theta Framework for formal verification and Java code for execution.

As it has already been stated in previous chapters, the language is designed to support formal verification methods by remaining within the boundaries of finite-state machines, but also to have as many high-level control and data structures, as this compromise allows. It extends the type system and expression language of the Gamma Statechart Composition Framework, but it is also extendable by tool-specific elements, which results in a seamless integration.

3.1 Elements of the Action Language

Actions - based on Definition 5 - are the fundamental unit of the behavior of the system. Depending on the type of the Action, it may take a set of inputs, produce a set of outputs, or modify the overall state of the system. Actions can be placed in specific places in a statechart: in states as entry or exit actions, or on transitions. They may have side-effects on variables and components that are located in the scope of the Action (data-flow) or influence the execution of other Actions (control-flow). Most actions are Statements, but there are also two special types of Actions: Blocks – that contain other actions – and Empty Actions – that represent no behavior. For an overview of the syntax of the complete action language, see Listing A.1;

The syntax of the language elements is given using the EBNF notation, described in [1]. Examples are given in the resulting language, the (high-level) action language of the Gamma Framework.

3.1.1 Value Declarations

Value types are an essential part of imperative programming languages. They pair a name with a data storage location, so the data can be accessed during later phases of the execution using the name. The data must have a predefined type, which is required for its unambigous handling and the pre-runtime discovery of errors. In the language,

there are currently two types of values that can be used as Actions: variable declaration statements and constant declaration statements. Another two, parameter declarations and field declarations – that are only placeholders for actual variables, only possessing a name and a type – are only accessible from other actions or expressions. For the list of types supported by the language and their semantics, see Table 3.1.

Type	Meaning
integer	integer number
bounded integer	integer number between predefined boundaries
boolean	true or false
enumeration	user-defined set of values
array	list of elements of a predefined type, accessed through a name and an integer-valued index
record	a set of values (fields) accessed through a name and a field name hierarchically
integer range	a set of integers between predefined boundaries

Table 3.1: The value types supported by the action language

Variable Declaration Statements are Statements that declare and optionally define the variables on which other Actions can operate. Variables must have a name, a type and may have a side-effect free expression for the definition of the variable. They can be read or written and are valid and unique in the scope in which they are declared. They are equivalent to variable declarations used elsewhere in the Gamma Framework. For the syntax and examples of the usage of variable declaration statements, see Listing 3.1.

```
VariableDeclarationStatement  = 'var' ID ':' Type [':=' Expression] ';' ;
```

```
var a : integer;
---
var b : boolean := true;
---
var c : array integer[3];
```

Listing 3.1: Syntax and examples of variable declaration statements

Constant Declaration Statements are Statements that declare and define read-only values, that can be used by other Actions. Constants possess a name, a type and a mandatory side-effect free expression for the definition of the constant. They are valid and unique in the scope in which they are declared. They are equivalent to constant declarations used elsewhere in the Gamma Framework. For the syntax and examples of the usage of constant declaration statements, see Listing 3.2.

```
ConstantDeclarationStatement  = 'const' ID ':' Type := Expression ';' ;
```

```
const a : integer := 5;
---
const b : array integer[2] := []{1, 2};
```

Listing 3.2: Syntax and examples of constant declaration statements

Parameter Declarations declare read-only values as parts of various other actions (i.e. they are placeholders for constants, possessing a name and a type). They are not independent elements of the action language, as the semantics of the value taken by the parameter depends on the containing action. For an example of the usage of parameters, see Listing 3.16.

Due to only specifying properties of identifiers (language entity namings), value declarations themselves do not have any effect on the formal verifiability of the correct behavior of the system. However, the values held by these declarations can be problematic, as unbounded integers or real numbers coming from the outside world may be impossible to verify. This problem is delegated to the tool performing the formal verification: a feasible way of tackling this problem may be through applying abstractions.

3.1.2 Expressions

Expressions combine values (or references to values), operators, and functions, and can be evaluated to a single value of a certain type. They may have side-effects (i.e. change the state of the system, like function access expressions), that justifies their applicability as Actions. Expressions can be evaluated to an empty expression having the *void* type, which cannot be used in other expressions. The applicable expression language is that of the Gamma Framework.

Reference Expressions are expressions that refer to (elements of) value declarations or procedure declarations. Generally, they can be used as parts of assignment statements or expression statements. Apart from 'simple' references to value declarations, there are four access expressions, which are specialized references:

- record access expression: used to access fields (elements) of a record through the name of the record and the name of the fields.
- array access expression: used to access elements or an array through the name of the array and an integer-type expression for the index of the element.
- function access expression: used to access a function (lambda or procedure). This expression may have side effects and cannot be assigned.
- select expression: used to non-deterministically access an element of an array, enumeration or integer range. This expression cannot be assigned.

```
//var arr : array integer[2] := []{1, 2};
//var a : integer;
a := arr[1];
---
//proc() is defined somewhere
//var b : boolean
b := proc();
```

Listing 3.3: Examples of reference expressions

Value Initializer Expressions are expressions as part of variable and constant delcarations. They cannot contain expressions with a side-effect, as they can also be used outside of actions – i.e. to define global variables in or even outside of a statechart. For examples of the usage of value initializer expressions, see Listing 3.4.

```
var a : integer := 5; //5 is an integer literal expression
---
var b : integer := a; //a is a reference expression
```

Listing 3.4: Examples of variable initializer expressions

Expression Statements are Statements that contain expressions. For the syntax and examples of the usage of expression statements, see Listing 3.5.

```
ExpressionStatement       = Expression ';' ;    //based on the expression language of Gamma
```

```
1;              //integer literal expression
---
true;           //boolean literal expression
---
-1;             //unary minus expression
---
[]{1, 2, 3}     //array literal expression
---
1 + 1;          //addition expression
---
5 div 2;        //integer division expression
---
1 < 2;          //less expression
---
true = true;    //equality expression
---
(1 + 1) * 2;    //primary expression ( ... )
---
//var a : integer
a;              //reference expression
---
myArray[2];     //array access expression
---
myFunction(1, true);  //function access expression
---
rec.fa;         //record access expression
```

Listing 3.5: Syntax and examples of reference expressions

The evaluation of a side-effect free expression over the permitted value types can be carried out in a finite number of steps and does not alter the state of the system, thus it does not have an effect on formal verifiability. There is currently one expression in the langue, which can produce side effects, namely the function access expression. Functions associate a name with an expression (side-effect free) or an action – in the latter case the formal verifiability depends on the formal verifiability of procedure declarations.

3.1.3 State Modification Actions

State modifying actions are also fundamental entities in imperative programming, as these actions describe the possible changes of the system components over time. These actions may modify different kinds of (inner) variables, or stimulate other systems or system components through events.

Assignment Statements are statements that enable the assignment of values to variables. The left-hand side of the assignment is a reference expression (referring to a variable declaration) and the right-hand side is an expression. After the assignment, the value of the variable referred on the left-hand side takes the evaluated value of the right-hand side expression. For the syntax and examples of the usage of assginment statements, see Listing 3.6.

As an assignment statement only changes the state of one variable, which can be carried out in one step, the possibility of formal verification is not hindered by assignment statements.

```
AssignmentStatement       = AssignableExpression ':=' Expression ';' ;
AssignableExpression      = ReferenceExpression | ArrayAccessExpression | RecordAccessExpression ;
```

```
//var myInt : integer;
myInt := -1;
---
//var myArr : array integer[2];
myArr := []{1, 2};
myArr[1] := 3;
---
//var myRec : record{field1 : integer, field2 : boolean} := (# field1 := 3, field2 := false #);
//procedure proc() : integer { return 4; }
myRec.field1 := proc();
```

Listing 3.6: Syntax and examples of assignment statements

Raise Event Actions are actions of the Gamma Framework, that extend the action language by enabling the raising of predefined events. Events are also constructs of the Gamma Framework, that, when raised, are sent through the specified port to the component that contains the port on the other end of the channel. Events can also have parameters, thus it is possible to send values to other components. For the syntax and examples of the usage of raise event actions, see Listing 3.7.

As a raise event action only changes the state of one channel, which can be carried out in a finite number of steps, the possibility of formal verification is not hindered by raising events.

```
RaiseEventAction    = 'raise' Port '.' Event '(' [Expression {',' Expression}] ')'  ;
```

```
//Toggle is a provided interface with the out event toggle
raise Toggle.toggle;
---
//Operator is a provided interface with the out event add(p : integer)
raise Operator.add(5);
```

Listing 3.7: Syntax and examples of raise event actions

Set Timeout Actions are actions of the Gamma Framework, that extend the action language with setting timeouts of the Gamma Framework to a provided value given in seconds or milliseconds. An internal timeout event is produced at the end of the specified time interval. For the syntax and examples of the usage of set timeout actions, see Listing 3.8.

As a set timeout action only changes the state of one timeout, which can be carried out in one step, the possibility of formal verification is not hindered by set timeout actions.

```
SetTimeoutAction      = 'set' TimeoutDeclaration ':=' TimeSpecification ;
TimeSpecification     = AdditiveExpression TimeUnit
TimeUnit              = 's' | 'ms'
```

```
//timeout timeOut
set timeOut := 5 s;
---
//timeout timeOut
//var offset : integer := 500
set timeOut := (100 + offset) ms;
```

Listing 3.8: Syntax and examples of set timeout actions

Deactivate Timeout Actions extend the action language with deactivating timeouts of the Gamma Framework. A deactivated timeout does not produce timeouts. For the syntax and examples of the usage of deactivate timeout actions, see Listing 3.9.

As a deactivate timeout action only changes the state of one timeout, which can be carried out in one step, the possibility of formal verification is not hindered by deactivate timeout actions.

```
DeactivateTimeoutAction   = 'deactivate' TimeoutDeclaration ;
```

```
//timeout timeOut
deactivate timeOut;
```

Listing 3.9: Syntax and examples of deactivate timeout actions

3.1.4 Control Flow Statements

Control flow statements explicitly define the order of execution for other actions. The chosen order of execution may depend on the overall state of the system (i.e. values), expressions, or other control flow statements.

Blocks are actions that define sequential execution for the contained actions and also limit the scope of other actions. They may contain or be contained in other actions (e.g. variable declarations). For the syntax and examples of the usage of blocks, see Listing 3.10.

Formal verification of the behavior contained in a block is possible when it is possible for each of the contained actions – which is detailed at the individual actions.

```
Block       = '{' {Action} '}' ;
```

```
{
  . . .
}
---
if(true) {
  . . .
}
```

Listing 3.10: Syntax and examples of blocks

If Statements are conditional statements that define blocks of actions that are executed when the guard expression of the block is evaluated to true. There can be an arbitrary number of guarded blocks, but at most one of them is executed. In case multiple guards are evaluated to true, the earlier defined one is chosen. There is also a possibility to define a block to execute when none of the other blocks is chosen, called an *else-block*. For the syntax and examples of the usage of if statements, see Listing 3.11.

The formal verification of the if statement is possible, if it is possible for all the contained actions, as *the guard expressions must always be side-effect free expressions.*

```
IfStatement     = 'if' '(' Expression ')' Action
  {'elsif' '(' Expression ')' Action}
  ['else' Action]
;
```

```
if (a /= b) {
  . . .
}
---
if (a < b) {
  . . .
```

```
  } elsif (a > b) {
    . . .
  } else {
    . . .
  }
```

Listing 3.11: Syntax and examples of if statements

Choice Statements are statements that define blocks of actions that are executed when the guard of the block is evaluated to true. There can be an arbitrary number of guarded blocks, but at most one of them is executed. In case multiple guards are evaluated to true, one block is chosen *non-deterministically*. For the syntax and examples of the usage of choice statements, see Listing 3.12.

The formal verification of the choice statement is possible, if it is possible for all the contained actions, as *the guard expressions must always be side-effect free expressions*. It is worth to note that this element may introduce non-deterministic behavior, which causes branching in the state-space.

```
  ChoiceStatement     = 'choice' '{'
    'branch' '[' Expression ']' Action
    {'branch' '[' Expression ']' Action}
    '}'
  ;
```

```
choice {
  branch [a < 1] {
    . . .
  }
  branch [a < 2] {
    . . .
  }
  branch [true] {
    . . .
  }
}
```

Listing 3.12: Syntax and examples of choice statements

Break Statements are statements that can be contained in *switch statements* and *for statements*, the resulting behavior depending on the type of the containing statement, in general interrupting the execution of the containing statement. For the syntax of break statements, see Listing 3.13. For examples of their usage, see Listings 3.15 and 3.16.

As break statements never increase the number of actions to be executed and do not change the state of the system in any way, they do not have an effect on the feasibility of the formal verification of the system.

```
  BreakStatement      = 'break' ';';
```

Listing 3.13: Syntax of break statements

Return Statements define the return values of *Procedures*, also interrupting their execution when encountered. For the syntax of return statements, see Listing 3.14. For examples of their usage, see Listing 3.18.

As return statements affect at most one value assignment, they do not have an effect on the feasibility of the formal verification of the system.

```
ReturnStatement       = 'return' Expression ';' ;
```

Listing 3.14: Syntax of return statements

Switch Statements are statements that define blocks of actions that are executed when the evaluation of the guard expression of the given or a previous block equals the evaluation of the control expression defined at the beginning of the statement. There can be an arbitrary number of guarded blocks, and the execution starts at the first block where the previously defined condition is met. The execution stops when a *break statement* is encountered. This means, that the language supports *fallthrough* switch statements. There is also a possibility to define a block whose guard is always evaluated to true, called a *default-block*, which must be the last of the blocks of the statement. For the syntax and examples of the usage of switch statements, see Listing 3.15.

The formal verification of the switch statement is possible, if it is possible for all the contained actions, as *the guard expressions and the control expression must always be side-effect free expressions.*

```
SwitchStatement     = 'switch' '(' Expression ')' '{'
    'case' Expression ':' Action
    {'case'  Expression ':' Action}
    ['default' ':' Action]
  '}'
;
```

```
switch (a) {
  case b : {
    . . .
  }
  case c : {
    . . .
    break;
  }
  case d : break;
  deafult: {
    . . .
  }
}
```

Listing 3.15: Syntax and examples of switch statements

For Statements are statements that define a *parameter variable* and a *body* (an action), and take a *range expression* that can be evaluated to a value of a composite, enumerable type (i.e. containing multiple values). The action is executed for each value contained in the range expression *sequentially*, with the parameter variable taking the corresponding value each time. The execution of the statement can be interrupted by a *break statement*. The for statement also has an action called the *then-action*, which is executed, if the execution of the for statement was not interrupted by break statements. For the syntax and examples of the usage of for statements, see Listing 3.16.

For statements are useful for the iterative solution of several well-known programming problems – such as raising a number to its powers or approximating logarithms –, as well as for the compact description of the same behavior taking different input parameters – like checking or behaving based on a series of previously saved inputs. Due to a range expression – of a previously known, fixed size – always being required at the definition of a for statement, the number of iterations is always known in advance. This prevents infinite loops, which would in turn make the termination of the program undecidable. Another

factor we must consider is verifiability of the body and then-actions, which is discussed at the individual actions.

The range expression is either an array literal expression, an integer literal expression, a reference to an array or integer range typed variable or a reference to an enumeration type definition. Due to the syntax and well-formedness constraints of these elements, these constructs always contain a finite set of elements, as either the elements or the size must be explicitly given by the designer of the system.

'Standard' for-loops of other imperative programming languages are not included in the language, as the iteration until an unbounded variable fulfilling certain criteria would result in possibly infinite loops thus livelocks, and also undecidability of the halting problem. However, through the inclusion of the integer range type into the action language, the behavior of these for-loops can be simulated in many cases, as often for loops have their parameter variable starting at a given value and increasing by steps of 1 until another given value. In case the for statement contains calculation boundaries resulting from the finiteness of the range variable, this can be signaled using the then-action (as in one of the examples of Listing 3.16).

```
ForStatement    =   'for' '(' ParameterDeclaration ':' Expression ')'
    Action
  'then'
    Action
;
ParameterDeclaration = ID ':' Type ;
```

```
//var myArr : array integer[3] := []{1, 2, 3};
for (p : myArr) {
  . . .
}
---
//Calculating a certain power of 'num' up to the 20th
//var num : integer := 10;
//var pow : integer := . . .;
//var res : integer := 1;
for (p : [1 .. 21]) {
  res := res * num;
  if(p >= pow)
    break;
} then {
  raise Events.error; //assuming the Events.error event exists
}
```

Listing 3.16: Syntax and examples of for statements

Assertion Statements are statements that offer monitoring capabilities at runtime. They contain a boolean-type expression, which, when evaluated to false, makes the assertion fail. On a failed assertion, the system stops operating. For the syntax and examples of the usage of assertion statements, see Listing 3.17.

Assertions are useful for setting requirements in addition to the trigger and guard of the transition. As they can occur anywhere in an action, more complex computations are possible for the requirement, than in the guard of the transition. They can also be used for monitoring the state of the system at runtime.

Due to assertions never increasing the number of actions to be executed and possibly causing the system to terminate, they do not have a negative effect on the feasibility of formal verification of the system.

```
AssertionStatement    =   'assert' '(' Expression ')' ';' ;
```

```
{
  //var pos : integer
  //pos := ... complex expression here ...
  assert(pos > 0);
}
```

Listing 3.17: Syntax and example of assertion statements

3.1.5 Other Elements

Empty Statements are special statements that do not represent any behavior. They can be used where actions are required, but there is no intended behavior. Empty statements do not have a textual representation and do not change properties of the action language (such as termination, syntax, etc.) in any way.

Procedure Declarations are language elements that define a reusable block of actions, which is executed at the corresponding *function access expressions*. They are not actions themselves, but contain an arbitrary number of actions through a block. For the syntax and examples of the usage of procedure declarations, see Listing 3.18.

Procedures (and functions in general) are useful, as they reduce the redundancy of the code, thus eliminating possible design flaws resulting from having to change the it in multiple places. Apart from their parameters, procedures of the language are independent from their context, which allows for inlining at the place of the access. This improves the verifiability by handling the code contained in the procedure together with its context, thus enabling further optimization.

An important factor to consider in connection with procedures, is the problem of recursion. If it is possible for a procedure to access itself and the return statement (or the end of the procedure) is never reached, the situation leads to a livelock – or assuming finite memory, some kind of non-deterministic memory overflow. Naturally, this leads to the impossibility of formal verification, thus it must be limited in some way. The current policy is to define the maximum recursion depth somewhere in the scope of the procedure declaration – the exact place depending on the target framework (the Gamma Framework).

As procedure declarations only consist of parameter declarations, a return value and a block of actions – also, the problem of recursion is resolved by limiting the execution of the elements to a previously known finite number – the feasibility of formal verification depends on the actions contained in the block – which is discussed at the individual actions.

```
FunctionDeclaration     = LambdaDeclaration | ProcedureDeclaration ;
LambdaDeclaration       = 'lambda' ID '(' [ParameterDeclaration {',' ParameterDeclaration}] ')'
  ':' Type ':=' [Expression] ;
ProcedureDeclaration    = 'procedure' ID '(' [ParameterDeclaration {',' ParameterDeclaration}] ')'
  ':' Type Block ;
```

```
procedure myProc() : void {
  . . .
}
---
procedure myProc2(a : integer, b : boolean) : integer {
  . . .
  return a + 5;
}
```

Listing 3.18: Syntax and examples of function declarations

3.2 Validation

Validation rules enforce the well-formedness constraints of the modeling language in the actual development environment. Some of these only warn the user of potential design flaws – so they can be discovered as early in the design phase, as possible – while others mark unintentionally non-deterministic or faulty behavior or an untransformable combination of Actions in the form of errors.

The exhaustive list of these validation rules is very long and consist of mostly intuitive elements. For this reason, most rules described here are only summarizing some aspects to be considered with respect to the corresponding priority (warning or error) instead of providing precise definitions of all the individual rules based on the metamodels of the language.

3.2.1 Warnings

- **Value Declarations:**
  - unused value declaration (variable, constant or parameter): it may cause overhead, although not resulting in undefined behavior
- **Expressions:**
  - side effect-free expression statement: it may cause overhead, although not resulting in undefined behavior
- **State Modification Actions:**
  - assignment of a variable to itself: it may cause overhead, although not resulting in undefined behavior
- **Control Flow Actions:**
  - empty block: it may cause overhead, although not resulting in undefined behavior
- **Other Elements:**
  - procedure only used once: it may cause overhead, although not resulting in undefined behavior

3.2.2 Errors

- **Value Declarations:**
  - constant declaration without initializing expression: it results in no value to be read
  - value declaration with different initializing expression type from the declaration type: type casting is not permitted in the language
  - void-type value: it makes no sense, as it cannot store anything
- **Expressions:**
  - invalid expressions (according to the constraints of the expression language)

  - initializing expressions having side effects: the execution would require actions, which may not be allowed in the given context
  - access expressions having invalid arguments (e.g. type, name, number of arguments): elements can only be accessed based on valid arguments

- **State Modification Actions:**
  - assignment to constant or parameter: by definition of these elements not allowed
  - assignment of different types: type casting is not permitted in the language
  - assignment of undefined values: it would result in an undefined behavior

- **Control Flow Actions:**
  - if, choice, switch statement guards (and control expressions) having side-effects: the state modifications should happen in the actions
  - return or break statements outside of any elements providing valid context for them: it would result in undefined behavior

- **Other Elements:**
  - procedure with different return statement types from each other and the type of the procedure itself: type casting is not permitted in the language

3.3 Low-level model

In order to transform our high-level models – that are easy to understand, handle and explain, with several language elements and their complex relations – into various other (often formal) models, it is useful to transform them into one with only a few different types of elements and many basic relations. For statecharts – like those of the Gamma Framework introduced in 2.6 –, this means substituting syntactic elements with ones resembling those of a finite-state machine: several transitions instead of hierarchical states or the product of the states of concurrent regions. These few simple constructs are then easily transformable into elements of other modeling languages.

For action languages, this means reducing the number of available actions, as well as expressions and types, by transforming the composite elements into constructs using only basic ones. Naturally, it is neither sensible nor possible to reduce the number of applicable elements indefinitely, but finding the elements resembling those of most other formalisms simplifies the further transformations greatly.

In the rest of the section, these high-level-to-low-level transformations of the language elements are examined. For the overview of the high-level-to-low-level transformations, see Table B.1. The example codes for each of the transformations are given through the Gamma Action Language syntax for the high-level statechart elements and the same syntax for the low-level statechart elements.

3.3.1 Value Declarations

The various value declarations of the high-level statechart model are transformed into variable declarations of the low-level model. The names and initializing expressions are easily transformable, the transformation of the types, however, is not so simple. The array

and record types – the complex types – arise from the composition of other, 'simple' types for more compact and often more intuitive handling. These types are broken up into their componenets, producing multiple declarations instead of the original one. For the list of type mappings, see Table 3.2.

High-level	Low-level
integer	integer
bounded integer	bounded integer
boolean	boolean
enumeration	enumeration
array	n × element type (also transformed)
record	the consituting types
integer range	n × integer

Table 3.2: The high-level-to-low-level type mappings

Variable Declaration Statements are transformed into one or more variable declaration statements, depending on the type of the original variable.

High-level	Low-level
`var a : integer;`	`var a : integer;`

`var b : array boolean[3] := []{true, true, false};`	`var __array_b_0 : boolean := true;`
	`var __array_b_1 : boolean := true;`
	`var __array_b_2 : boolean := false;`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.3: Examples of variable declaration statement transformations

Constant Declaration Statements are transformed into one or more variable declaration statements, depending on the type of the original constant.

High-level	Low-level
`const a : boolean;`	`var a : boolean;`
`const b : record{ f1 : boolean, f2 : integer} := (# f1 := true, f2 := -1#)`	`var __record_b_f1 : boolean := true;` `var __record_b_f2 : integer := true;`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.4: Examples of constant declaration statement transformations

Parameter Declarations are transformed into one or more variable declaration statements and one or more assignment statements, depending on the type and context of the original parameter. As they are not independent elements of the action language, the exact place and value assignment of the variable depends on the language construct the parameter was contained in, e.g. for statement or procedure declaration.

High-level	Low-level

`for (p : integer in ExampleArray) {`
`  . . .`
`}`
	`var p : integer;`
	`p := . . .`
	`. . .`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.5: Examples of parameter declaration transformations

3.3.2 Expressions

The result of the high-level-to-low-level transformation in case of expressions depends on the type and context of expressions. The unnecessary expressions – that do not affect the semantics of the model – are removed, the others are retained as parts of actions or are completely transformed into actions. Reference expressions are replaced with reference expressions referring to the corresponding variables of the low-level model.

Expression Statements are either replaced with empty actions (if they are side-effect free), or become a series of actions, like procedure declarations. Procedure declarations are inlined with the constraints of the inlining discussed in 3.1.

High-level	Low-level
`5 + 4;`	`//Empty action, as the value is not used`

`//procedure declared somewhere`
`proc();`
	`var __ret_proc : integer; //if not void`
	`{`
	`  . . . //maybe raise event, etc.`
	`}`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.6: Examples of expression statement transformations

Reference expressions are transformed based on the type of the value they refer to (and possibly access the values of) and the action they are used in. Generally, the resulting models declare a variable before the rest of the containing action, assign this variable some value, then replace the corresponding part of the action with a reference to this variable. In case of 'simple' reference expressions, the high-level reference can simply be replaced, as this intermediate assignment can be omitted.

- In case of **array access expressions**, the variable is assigned based on the runtime evaluation of the argument expression, within the boundaries of the defined array.
- In case of **function access expressions**, the function is inlined before the assignment and the return variable of the function is assigned. The details and difficulties of this inlining are discussed in Section 3.1 at the procedure declarations.
- In case of **record access expressions** the generated variable can be directly assigned, as the name of the accessed record and field is known at compile time (at the time of the model generation).

- In case of **select expressions**, the capabilities of choice statements are utilized for non-deterministic value assignment.

Value Initializer Expressions can be copied into the low-level model, as action language-specific expressions are not allowed.

3.3.3 State Modification Actions

State modification actions are transformed into assignment statements, as the different aspects of the overall state of the system are stored in variables.

Assignment Statements are transformed into assignment statements of the low-level model. The expressions on the left and righ-hand sides of the assignment are transformed according to the transformation rules of expressions.

High-level	Low-level
`//variable 'a' is valid` `a := 3;`	`//transformation of variable 'a' exists` `a := 3;`

`//variables 'a' and 'myRec' are valid`
`a := myRec.f1;`
	`//transformations of the variables exist`
	`var __rhs_val : integer;`
	`__rhs_val := __record_myRec_f1;`
	`a := __rhs_val;`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.7: Examples of assignment statement transformations

Raise Event Actions are transformed into one or more assignment statements. In case of unparametrized events, it is enough to assign the corresponding *isRaised-flag*, but for parametrized events, each of the parameters must be assigned, if the corresponding arguments exist.

High-level	Low-level
`//event 'Out.out' is valid` `raise Out.out;`	`//event is transformed` `isRaised := true; //out.isRaised variable`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.8: Examples of raise event action transformations

Set Timeout Actions are transformed into an assignment statement. The left-hand side of this assignment is the low-level timeout variable (corresponding to the high-level timeout declaration) and the right-hand side is an integer literal, setting the timeout variable to 0 – which is then supposed to count from 0 to infinity (in practice up to the given time).

Decativate Timeout Actions – being Gamma Framework-specific actions – require further consideration and are not yet transformed.

3.3.4 Control Flow Actions

Control flow actions are transformed into either blocks (if they represent sequential execution), if or choice statements (if they represent choice between executions, depending on the fact if they represent deterministic choice or not), or are removed from the resulting model, but influence the contents of other elements.

Blocks are transformed Blocks of the low-level models. Each of the contained actions of the original block is transformed and placed in the block resulting from the transformation.

If Statements are transformed into if statements. The guards are extracted into separate boolean-valued variables (thus supporting the resolution of access expressions), which then replace the original guards. The branches of the if statement contain the transformed images of the original actions.

To be able to support the transformation of for statements, switch statements and procedure declarations – that can have break and return statements as elements –, the actions after the statement are placed in each of the branches and an else branch is also defined in case there is none.

High-level	Low-level

`//events are declared`
`//G1 is a boolean-type expression`
`if (G1) {`
`  raise B;`
`}`
`raise C;`
	`//events are transformed`
	`//G1 is the same expression`
	`var __g1 : boolean;`
	`__g1 := G1  //maybe transformed further`
	`if (g1) {`
	`  raise B;`
	`  raise C;`
	`}`
	`else {`
	`  raise C;`
	`}`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.9: Examples of if statement transformations

Choice Statements are transformed into choice statements. The guards are extracted into separate local boolean-valued variables (thus supporting access expressions), which then replace the original guards. The branches of the choice statement contain the transformed images of the original actions.

To be able to support the transformation of for statements, switch statements and procedure declarations – that can have break and return statements as elements –, the actions after the statement are placed in each of the branches, and also an extra if statement is added, similarly to the extra else branch at the transformation of the if statement, which is only supposed to run when none of the branches of the choice statement could. For this reason, an additional local flag-variable must be added to the result of the transformation.

Break Statements do not have any images in the transformed models. They simply affect the contents of the the choices (if and choice statements) that are the results of transformations of for and switch statements. For an example of break statements on the low-level models, see Table 3.12;

High-level	Low-level

`//events are declared`
`//G1 and G2 are boolean-type expressions`
`choice {`
`  branch [G1] raise A;`
`  branch [G2] raise B;`
`}`
`raise C;`
	`//events are transformed`
	`//G1 and G2 are also transformed`
	`var __g1 : boolean;`
	`__g1 := G1; //maybe transformed further`
	`var __g2 : boolean;`
	`__g2 := G2; //maybe transformed further`
	`var __flag : boolean := false;`
	`choice {`
	`  branch [__g1] {`
	`  __flag := true;`
	`  raise A;`
	`  raise C;`
	`  }`
	`  branch [__g2] {`
	`  __flag := true;`
	`  raise B;`
	`  raise C;`
	`  }`
	`}`
	`if (__flag) {`
	`  raise C;`
	`}`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.10: Examples of choice statement transformations

Return Statements are transformed into assignment statements in the low-level models, if the type of the containing procedure is non-void, or simply have an effect on the choices (if and choice statements) that are the results of transformations of the corresponding procedure declarations. For examples of return statement transformations (along with the containing procedure), see Table 3.14.

Switch Statements are transformed into if statements. For the control variable and each of the guards a variable is declared and assigned (to be able to support access expressions), then an if statement is created, which has guards comparing these variables, and bodies that contain the actions of the body of the switch-branch of the same index and *all subsequent switch-branches* (fall-through). The default branch is transformed into an else branch, if exists, or a default branch is defined when none exists. As a break statement can occur anywhere inside a switch statement, which determines the occurrence and execution of other language elements, the rest of the execution is explicitly defined for each case using additional if statements.

High-level	Low-level

`//variables a, b defined`
`//events declared`
`switch (5) {`
`  case a : {`
`  raise A;`
`  }`
`  case b : {`
`  raise B;`
`  break;`
`  }`
`  default : {`
`  raise C;`
`  }`
`}`
`raise D;`
	`//the variable and event images exist`
	`var __control : integer;`
	`__control := 5;`
	`var __guard1 : integer;`
	`__guard1 := a;`
	`var __guard2 : integer;`
	`__guard2 := b;`
	`if (__guard1 = __control) {`
	`  raise A;`
	`  raise B;`
	`  raise D;`
	`}`
	`else if (__guard2 = __control) {`
	`  raise B;`
	`  raise D;`
	`}`
	`else if (true) {`
	`  raise C;`
	`  raise D;`
	`}`
	`else if (true) {`
	`  raise D;`
	`}`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.11: Examples of switch statement transformations

For Statements are *unrolled* in the low-level models, i.e. for each parameter value, the parameter variable is assigned to the corresponding value – this means a generated variable declaration beforehand – and the body of the statement is executed under this precondition. The contents of this unrolled for statement are then transformed according to the other transformation rules. As a break statement can occur anywhere inside a for statement, which determines the occurrence and execution of other language elements, the execution is explicitly defined for each contained (deterministic or non-deterministic) choice using if and choice statements.

High-level	Low-level

`//variable myArray is []{1, 2}`
`//variable cnt is declared`
`for (p : integer in myArray) {`
`  cnt := cnt + 1;`
`}`
	`//the variable transformations exist`
	`var p : integer;`
	`p := 1; //first array element is 1`
	`{ cnt := cnt + 1; }`
	`p := 2; //second array element is 2`
	`{ cnt := cnt + 1; }`

`//variables and events declared`
`for (p : integer in [0 .. 1]) {`
`  raise A`
`  if (p = 1) {`
`  raise B;`
`  break;`
`  }`
`  raise C;`
`}`
`then {`
`  raise D;`
`}`
	`//the variable and event transformations`
	`//exist`
	`var p : integer;`
	`p := 0;`
	`raise A;`
	`if (p = 1) {`
	`  raise B;`
	`}`
	`else {`
	`  raise C;`
	`  p := 1;  //start of next inline pass`
	`  raise A;`
	`  if (p = 1) {`
	`  raise B;`
	`  }`
	`  else {`
	`  raise C;`
	`  //No more parameter values -> then`
	`  raise D;`
	`  }`
	`}`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.12: Examples of for statement transformations

Assertion Statements are transformed to assertion statements. The contained expression is also transformed by extracting it into a separate, temporary variable (to support access expressions). In the low-level assertion statement, a reference to this variable is going to be the assertion expression.

High-level	Low-level

`//G1 is a boolean-type expression`
`assert(G1);`
	`//G1 is transformed`
	`var __g1 : boolean;`
	`__g1 := G1; //maybe transformed further`
	`assert(__g1);`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.13: Examples of assertion statement transformations

3.3.5 Other Elements

Empty Statements are transformed into empty statements. They represent no behavior in the low-level models either, but are the result of the transformation of several other

language elements. For an example of transformations resulting in empty statements, see Table 3.6;

Procedure Declarations are *inlined* into the transition where the procedure was called from. This inlining is possible, as the problem of infinite recursion was already addressed. In case the procedure is of non-void type, a return variable is declared. The (images of the) parameters are assigned the corresponding arguments of the calling expression. The contents of this inlined procedure are then transformed according to the other transformation rules. As a return statement – which assigns a value to the return variable and ends the execution of the actions inside the procedure – can occur anywhere inside a procedure declaration, which determines the occurrence and execution of other language elements, the execution is explicitly defined for each possible choice using if and choice statements.

An example for the transformation of procedure declarations can be seen in Table 3.14. Another example is in Table 3.6;

High-level	Low-level

`//place of calling:`
`var a : integer;`
`a := proc(2);`
 	
`//the procedure declaration:`
`procedure proc(param : integer) : integer`
`{`
`  return param * 2;`
`}`
	`//image of the place of calling:`
	`var a : integer;`
	`//the inlined procedure:`
	`var __proc_ret : integer;`
	`var __proc_param1 : integer;`
	`__proc_param1 := 2;`
	`{`
	`  __proc_ret := __proc_param1 * 2;`
	`}`
	`//`
	`a := __proc_ret;`

NOTE: the names of low-level variables are automatically generated and may be different

Table 3.14: Example of a procedure declaration transformation

3.4 XSTS

For the formal verification of the designed systems, currently the transformation to the experimental eXtended Symbolic Transition System (xSTS) formalism of the Theta Framework [19] is supported. This formalism contains very low-level constructs: only variables, assignments, assumptions and conditional and parallel branching of the control flow are supported, with an expression language only supporting basic mathematical and logic expressions. This formalism can be easily transformed into SMT formulas, which are used to describe the state changes in the Theta Framework.

3.4.1 XSTS syntax

The xSTS formalism contains the following elements:

Expressions are similar to the side-effect free expressions used in the action language, as they are both based on the same metamodel. Compared to the expressions used in the action language, there are no access expressions (as complex data types are not allowed), and cannot have void as type.

Assume Actions state assumptions about the state of the system at a given point during the execution. They contain boolean-type expressions, which, when evaluated to false, make the assumption fail. On a failed assumption, the whole execution of the action containing the assumption fails and the state modifications up to the assumption are also canceled.

Empty Actions represent no behavior and may be freely removed. They are useful during the transformation of the models.

Assignment Actions represent the same behavior as assignment statements of the action language. They assign the result of the evaluation of the right-hand side expression to a variable given with a reference expression on the left-hand side.

Sequential Actions define sequential execution for the contained actions. In this regard, they are similar to blocks of the action language.

Non-deterministic Actions define a non-deterministic choice between their contained actions.

Parallel Actions define parallel execution for their contained actions.

Transformation of the low-level action language elements:

In the rest of the section, these low-level-to-xSTS transformations of the language elements are examined. For an overview of the low-level statechart-to-xSTS transformations, see Table B.2.

The example codes for each of the transformations are given through the Gamma action language syntax for the low-level statechart elements and the xSTS syntax for the xSTS elements.

3.4.2 Variable Declarations

In the low-level statechart model, only variable declarations are allowed as images of high-level action language elements. These variable declarations need to be transformed into variables of the xSTS models. This transformation is very simple, as the names, types and initializing expressions can take the same values, as those of the low-level model.

Variable Declaration Statements are transformed into temporary variable declarations of the xSTS formalism, and are replaced with empty actions at the image of the place of the declaration.

High-level	Low-level

`{`
`  . . .`
`  var a : integer := 5;`
`  . . .`
`}`
	`var _a : integer = 5; //Global variable`
	`---`
	`{`
	`  . . .`
	`  //Empty action, later maybe removed`
	`  . . .`
	`}`

NOTE: the names of the xSTS variables are automatically generated and may be different

Table 3.15: Example of a variable declaration statement transformation

3.4.3 Expressions

Expressions of low-level statecharts are transformed into expressions of xSTS models. It can be assumed that the low-level model does not contain any expressions requiring the use of actions (e.g. access expressions), thus, they can be easily transformed to xSTS expressions.

Reference Expressions are transformed into xSTS reference expressions by creating a reference expression referring to the image of the originally referred variable.

Variable Initializing Expressions can be copied into the xSTS model, as every possible element corresponds to a similar element of the xSTS model.

3.4.4 State Modification Actions

Assignment Statements are transformed into the assignment actions of xSTS models. The expressions on the left-hand side and right-hand side of the action are transformed according to the rules of low-level statechart-to-xSTS expression transformations discussed above.

3.4.5 Control Flow Statements

Blocks are transformed into the sequential actions of the xSTS formalism. These sequential actions – similarly to blocks – define seqential execution for the contained actions.

If Statements are transformed into non-deterministic choices (actions) and assume actions in xSTS models. Each branch of the non-deterministic action contains all images of the actions of the corresponding if-branch and is guarded with its own guard conjoint with the negated guards of all the previous branches.

High-level	Low-level

`//G1 is a boolean-type expression`
`//'a' and 'b' are defined`
`if (G1) {`
`  a := b;`
`}`
	`//G1, a and b transformed`
	`choice {`
	`  assume (G1);`
	`  a := b;`
	`}`

`//G1, G2 and G3 are boolean-type`
`//expressions, 'a' is declared`
`if (G1) {`
`  a := 1;`
`}`
`else if (G2) {`
`  a := 2;`
`}`
`else {`
`  a := 3;`
`}`
	`//G1, G2 and G2 are transformed`
	`//'a' is transformed`
	`choice {`
	`  assume (G1);`
	`  a := 1;`
	`} or {`
	`  assume (!G1 && G2);`
	`  a := 2;`
	`} or {`
	`  assume (!G1 && !G2 && true);`
	`  a := 3;`
	`}`

NOTE: the names of the xSTS variables are automatically generated and may be different

Table 3.16: Example of an if statement transformation

Choice Statements are transformed into non-deterministic choices (actions) and assume actions of the xSTS formalism. The actions of the non-deterministic choice correspond to the branches of the choice statement and the assume actions arise from the transformation of the guards of its branches.

High-level	Low-level

`//G1, G2 and G3 are boolean-type`
`//expressions, 'a' is declared`
`choice {`
`  branch [G1] a := 1;`
`  branch [G2] a := 2;`
`  branch [G2] a := 3;`
`}`
	`//G1, G2 and G2 are transformed`
	`//'a' is transformed`
	`choice {`
	`  assume (G1);`
	`  a := 1;`
	`} or {`
	`  assume (G2);`
	`  a := 2;`
	`} or {`
	`  assume (G3);`
	`  a := 3;`
	`}`

NOTE: the names of the xSTS variables are automatically generated and may be different

Table 3.17: Example of a choice statement transformation

Assertion Statements are transformed into assume actions of the xSTS formalism. As the syntax and semantics of the assertion statement are similar to the syntax and semantics of the assume action, only the contained expression needs further transformation.

High-level	Low-level
`//G1 is an expression` `assert(G1);`	`//G1 is transformed` `assume(G1);`

Table 3.18: Example of an assume statement transformation

3.4.6 Other Elements

Empty Actions are transformed into empty actions of the xSTS formalism. Empty actions are similar to empty statements, they do not represent any behavior and can be removed.

3.5 Executable code

Java code is generated using the xSTS and the high-level statechart models, according to object-oriented principles. For every component, three Java classes are generated: an *interface-class*, which contains the elements on the interface of the statechart (which are publicly accessible from the outside world), a *statemachine-class*, which contains the logic in the statechart using low-level elements, and a *statechart-class*, which is a wrapper for the statemachine according to the elements of the original statechart model. The statemachine class is generated based on the xSTS model, the statechart class and the interfaces are generated based on the high-level statechart models.

The details of the Java code generation are omitted here, as it is entirely done by the Gamma Framework. It is capable of the transformation of the behavior without having to explicitly define the transformation of the action language elements due to the behavior being extracted from xSTS actions, which are independent from the action language proposed in this work – but are the targets of their transformations. For this reason, the details of this Gamma statechart-to-Java transformation are not discussed here, for that we refer the reader to [9, 18].

Chapter 4

Implementation

In this chapter, the implementation details of the action language are discussed. It consists of two parts: in Section 4.1 the tools and frameworks are presented, the new module builds upon. In Section 4.2 the integration into the Gamma Framework is discussed, from two different aspects: how the language is integrated into the framework, and how the transformation of the language elements is integrated into the other transformations of the framework.

4.1 Technologies

The applied technologies during the development of the action language correspond to the technologies applied during the development of the Gamma Framework. These are mostly open-source technologies: the target environment is the Eclipse platform, with the Eclipse Modeling Framework (EMF) providing the metamodel specification environment. The Xtext framework was used for the development of the modeling language. The majority of the model transformations was implemented using the Xtend programming language. The complete list can be found in [9], here only the action language-relevant technologies are presented.

4.1.1 Eclipse Environment

Eclipse is a popular, open-source integrated development environment (IDE). It is mainly used for Java-related application development, but also supports several other programming languages. It consists of a base workspace and an extensible plug-in system. Using this plug-in system, the develpment environment is easily customizable for different purposes, such as programming in different programming lanugages, modeling (using the Gamma Framework or Yakindu), or testing.

Eclipse Modeling Framework: The Eclipse Modeling Framework is an Eclipse-based modeling framework and code generation facility. It defines its own structured data model – called Ecore – for describing models and providing runtime support for the models. Models are defined using the XML Metadata Interchange (XMI) format, which is supported by various Eclipse plugins developed specifically for this purpose, as EMF is fully integrated into the Eclipse platform. It provides an environment to numerous technologies, including server solutions, persistence frameworks, UI and transformation frameworks.

4.1.2 Xtext Framework

Xtext is an open-source framework for developing (mostly) domain-specific languages (DSLs). It has its own syntax for the definition of textual languages, resembling a context-free grammar extended with mappings to the in-memory representations. Unlike standard parser generators, it generates not only a parser, but also the abstract syntax tree (AST) of the grammar, and also support several other features, such as validation rules and editing support. This is because Xtext is based on the EMF project – the metamodels of the defined languages are Ecore models –, and it is integrated into the Eclipse environment.

Xtend Xtend is a general-purpose, high-level programming language based on Java. It is statically typed, object-oriented and uses the type system of Java. Xtend programs are compiled to Java code, thus allowing seamless integration with existing Java libraries. It provides numerous convenient extensions to Java, such as dispatch methods, type inference, operator overloading and extension methods.

4.2 Gamma Integration

Most of the elements of the action language described in Chapter 3 are already integrated into the Gamma Framework. This integration is twofold: the high-level action language itself – implemented using the Xtext framework – had to be integrated into the statechart language, and the the transformation of the statechart models had to be extended with the transformations of the new elements.

4.2.1 Integration of the High-level Action Language

In the Gamma Framework, the statechart language consisted of two parts: the expression language (previously known as constraint language), which also contained the type language, and the statechart language, which contained the elements of statecharts. These statechart elements depended on elements of the expression language. The integration at this level was simple, as only a new element – the action language – had to be inserted into the dependency chain. It might be worth to note that these parts each consisted of two eclipse plugins: one for the Ecore metamodel and one for the Xtext grammar and the related functionalities. The resulting plugin dependencies can be seen on Figure 4.1.

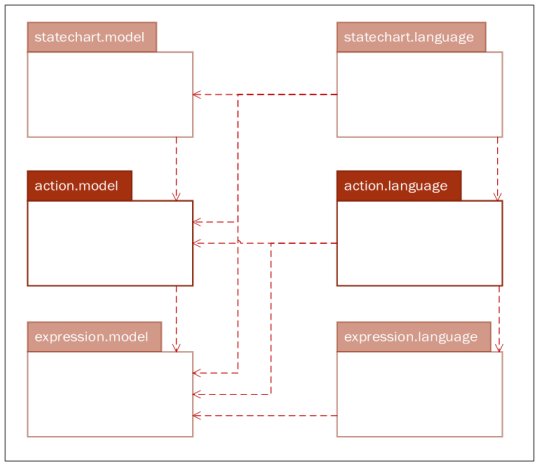


Figure 4.1: Dependencies of the high-level statechart plugins

4.2.2 Integration of the Transformations

For the integration of the action language into the transformations, the existing transformation plugins had to be extended. Even though there were no separate plugins for the transformation of different parts of the statecharts, the dependencies of the classes inside the transformation plugins resembled those of the high-level plugins, thus they were easy to extend. The high-level schematic description of the implementation of the model transformations can be seen on Figure 4.2.

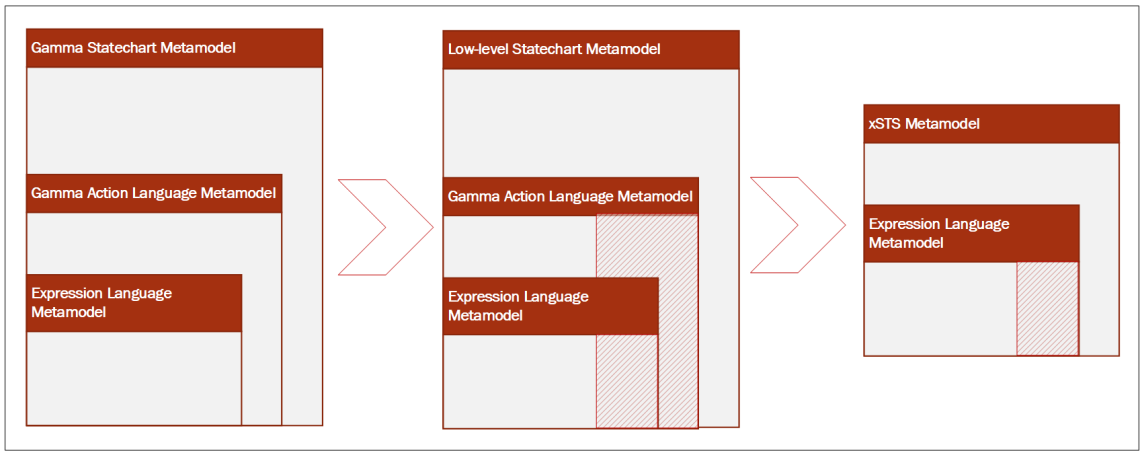


Figure 4.2: Dependencies of the high-level statechart plugins

The transformation of access expressions:

Access expressions – although not directly part of the action language, but through the expression language – can only be resolved using actions, due to the current capabilities of the rest of the framework. This resolution happens in the high-level-to-low-level transformation. This means, that access expressions can only be permitted in places where actions can be used too. For this reason, access expressions cannot be used in e.g. initializer expressions of various value declarations. Also, as the resolution of access expressions requires the use of actions, these additional actions must be inserted at the place where these expressions are used. The resolution of the individual access expressions depends on their concrete type.

- record access expressions can be resolved relatively easily. As records group individual variables (called fields) under a common name, with the fields also having names, the corresponding variable can be found using these informations, which are explicitly defined at compile time.
- array access expressions impose a great overhead on the system. As the index can be given using any kind of integer-type expression, the current means for the resolution of array access expressions is declaring a temporary variable which is assigned in an if statement that contains a branch for every possible value this index can take.
- select expressions are resolved similarly to array access expressions, with the choice being made in a choice statement, with branches guarded by true expressions, thus completely non-deterministically.
- function access expressions are resolved as described in Sections 3.1 and 3.3 – by inlining the accessed procedure (as lambdas are not yet supported).

Chapter 5

Results

This chapter presents the applicability of the action language. In Section 5.1, the testing methods used to validate the elements of the action language are presented. Then, in Section 5.2, a case study of a well-known programming problem is discussed. This case study is about modeling a calculator able to parse arithmetic expressions given by the postfix notation, also applying the newly introduced action language.

5.1 Validation of the Language Elements

The action language presented in this work is very complex, as it extends and is also extended by the Gamma Framework. For this reason, it is essential to analyze the behavior of the individual elements, especially after the model transformations.

It is possible to unit test the construction of the AST from the textual syntax, however, the Eclipse platform and the Xtext Framework provide an excellent opportunity to observe its structure through the *Outline* view anytime during the modeling. Naturally, this is not the best solution for proving the (in)correctness of the parsing and finding the design flaws, but, as the Xtext grammar is relatively simple and compact, the priority of having to test this module is significantly lower.

It would be beneficial to test the correctness of the high-level-to-low-level transformations. At this phase, the only option is to analyze the Ecore model resulting from the transformations, as there is currently no serialization option for the low-level models apart from XMI. However, the manual creation of Ecore models is very inefficient and slow and makes the test cases unreadable (not to mention mocking these objects). The low-level-to-xSTS transformations should also be analyzed, but the problem is the same as at the high-level-to-low-level transformations: it is a M2M transformation applied to Ecore models. The last step in this pipeline is the xSTS-to-Java codegeneration along with the statechart-to-Java codegeneration. In this case, both parts of the model transformation should be verified. This makes creating a test suite extensive enough to verify the correctness of all the individual transformations of each of the individual language elements and their most common combinations impossible.

Testing the entire workflow: the proposed alternative is provide the Gamma Framework with simple models containing the most common combinations of action language elements, then create test suites using Java – also exploiting the capabilities of JUnit or a similar unit testing framework – to test the resulting Java classes against the intended

behavior of the models. The schematic statechart of the proposed test models can be seen on Figure 5.1.

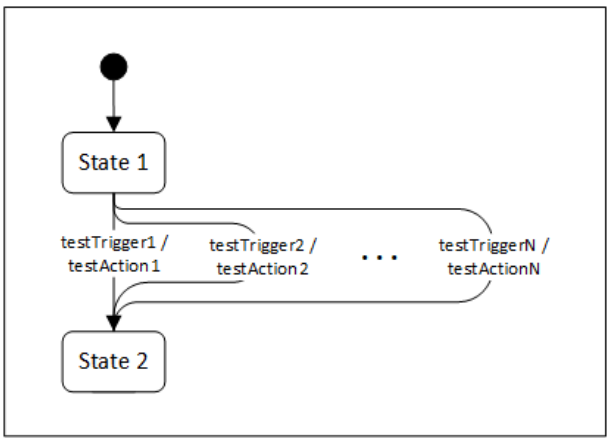


Figure 5.1: Schematic statechart of the proposed validation models

During the testing of the individual language elements, one such statechart was implemented for each of the main constructs. To each of these statecharts belogs a Java test class, with one test case testing one of the transitions of the statechart against its expected behavior. An example of one such test case can be seen in Table 5.1.

It is important to note, that tests implemented this way are black-box tests – i.e. they only test the functionality, not the implementation of the tested elements for two reasons. Firstly, they do not test the individual steps of the transformation workflow. Secondly, only the public interfaces of the generated Java classes are tested, which means, that we can only test event raisings directly (and the other elements only through events). On one hand, this is beneficial, as we are able to gain information on the (in)correctness of the individual language elements despite the number of tests remaining relatively small. On the other hand, whenever a tested functionality fails, there is no information provided about the cause of the problem, thus the correction of the code requires significantly higher effort.

The results of the validation of the individual language elements at the time of the writing this thesis are summarized in Table C.1.

Functionality to test	The corresponding test case

`//To test: in case of two branches of`
`//an if statement are executable,`
`//the earlier defined one is chosen`
 	
`transition from state1 to state2`
`  when Input.ifG1elsifG2TrueLit / {`
`  raise Output.pre;`
 	
`  if (true) {`
`  raise Output.ifG1elsifG2_A;`
`  }`
`  elsif (true) {`
`  raise Output.ifG1elsifG2_B;`
`  }`
 	
`  raise Output.ifG1elsifG2_C;`
`}`
	`@Test`
	`void`
	`testIfStatementStatechartIFG1elsifG2TrueLit`
	`() {`
	`  // Arrange`
	`  IfStatementStatechart sc =`
	`  new IfStatementStatechart();`
	`  IfStatementStatechartInterface`
	`  scInterface = sc;`
	
	`  // Act`
	`  scInterface.reset();`
	
	`  scInterface.getInput()`
	`  .raiseIfG1elsifG2TrueLit();;`
	`  scInterface.runCycle();`
	
	`  // Assert`
	`  Assertions.assertTrue(sc.getOutput()`
	`  .isRaisedPre());`
	`  Assertions.assertTrue(sc.getOutput()`
	`  .isRaisedIfG1elsifG2_A());`
	`  Assertions.assertFalse(sc.getOutput()`
	`  .isRaisedIfG1elsifG2_B());`
	`  Assertions.assertTrue(sc.getOutput()`
	`  .isRaisedIfG1elsifG2_C());`
	`}`

Table 5.1: Testing the execution of the correct branch of an if statement

5.2 Case study: RPN Calculator

This section demonstrates the capabilities the action language. It presents a problem commonly solved in general-purpose imperative programming languages, offers a possible solution using statecharts of the Gamma Framework – extended with the action language described in this thesis – and evaluates the generated executable and verifiable code.

5.2.1 Introduction

The problem to solve is the parsing and evaluation of a mathematical expression that can contain integer numbers and basic arithmetical symbols (+, -, *, /). For the input to be valid for calculation, the expression must be a valid mathematical formula given with reverse polish (postfix) notation [5]. The calculator should be able to check the validity of the formula and signal an error in case it was invalid, or evaluate the formula, return the result and signal success if it was valid.

It would also be worthwhile to design a tester component in addition to the calculator, which generates a valid input sequence and feeds it to the calculator on its input port. This would allow the users to observe the behavior of the calculator without having to produce the input themselves.

5.2.2 Designing the system

The ports of the system: To implement the above described system, we require an interface, through which we can signal the tester to start feeding input to the calculator. We also wish to observe the result of the calculation, for which we provide an interface to the system. Our input should be sent through an interface provided by the tester, and the output should arrive on an interface provided to the calculator. The tester should be connected to the calculator through the input interface provided by the calculator and required by the tester.

The input interface of the system (and also the tester), *TesterInput*, only needs one type of event to *start* the operation. The output interface of the system (also the calculator) can signal two different events: *success* and *error*. The interface of the calculator, *CalcInput*, can take three different inputs: a single *operator*, a single *operand*, or the *evaluate* command.

The components of the system and their connections are summarized on Figure 5.2.

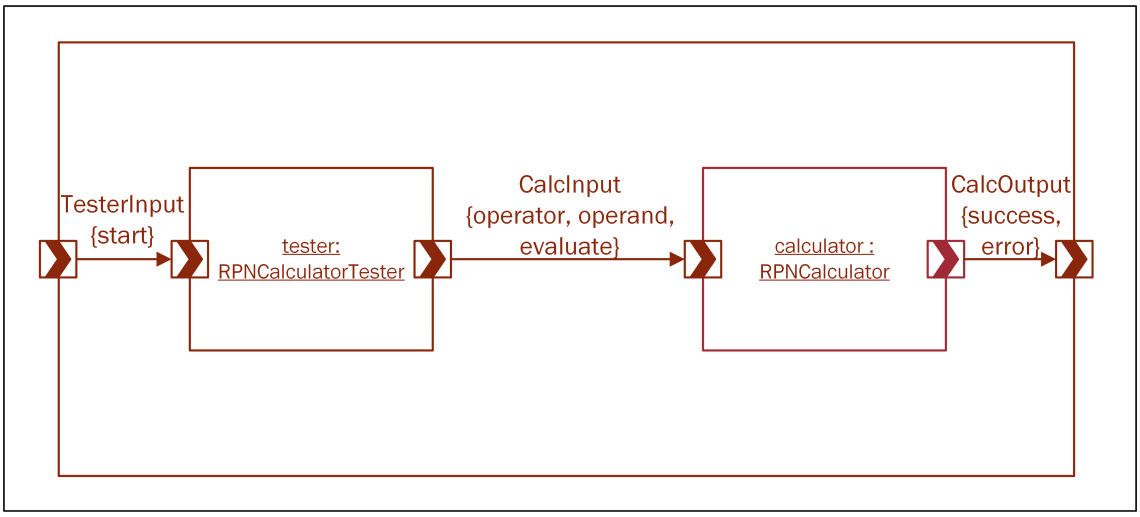


Figure 5.2: Component diagram of the modeled system

The states of the calculator:

A formula is valid according to the rules of postfix notation, if it starts with two operands (or contains only one operand altogether), ends with an operator, and contains one less operators than operands – as each possible operator takes two operands. As we are modeling the system using statecharts, the states of the system present a convenient way to verify the validity of the given expressions. The calculator should start in an initial state (*'Init'*), which represents a state where the list of inputs is empty. As one number is considered a valid expression, there should be a state (*'OneNumber'*) representing the validity of expressions consisting of only one operand. In this case, the logic of the evaluation is simple, as only that number needs to be returned. Apart from this special case, the expression should start with two numbers, but two operands without an operator are considered invalid (*'Invalid'*). In the later phases, the system should return to this state, whenever the expression ends with an operand, as the expression should end with an operator. If the expression ends with an operator, the formula *may* be considered valid (*'MaybeValid'*), as the determination of the validity requires further computation.

The transitions of the calculator:

In each possible state of the system, a transition should be defined for each of the possible inputs, as every input should result in some kind of behavior (even as simple as saving the input). In general, there are three kinds of different behaviors: when the formula is invalid and cannot become valid anymore, or when the evaluation of an invalid formula is required, the system should return to its initial state, clearing the previously saved data, and send an error on the output interface. When the formula may at one point in the future be considered valid, the input should be saved and the states should represent the current validity. When the formula is valid and an evaluation is required, the system should calculate and return the result and clear the stored input. The statechart representation of the calculator can be seen on Figure 5.3.

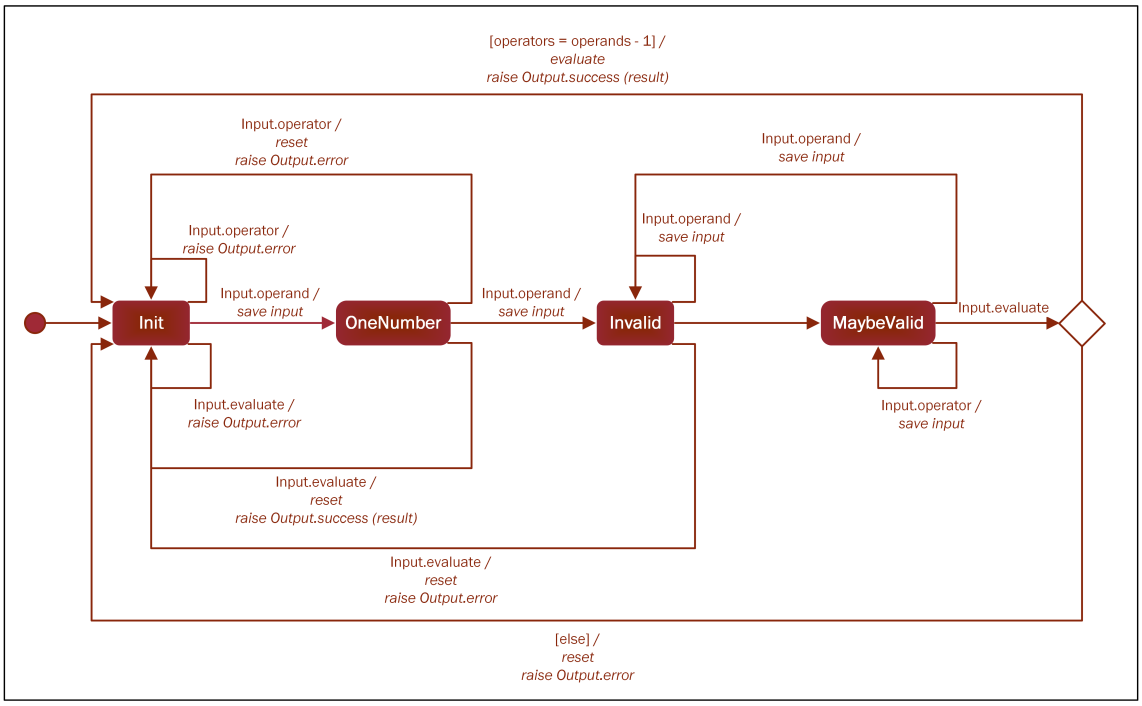


Figure 5.3: Statechart of the calculator

The tester:

The structure and behavior of the tester is very simple, it only consists of two states, one possible input event, one timeout event and three possible output events. The statechart representation of the tester can be seen on Figure 5.4.

On the figures, the actions have been marked with *italics*.

5.2.3 The Gamma Model

The details of implementing the statecharts and their interfaces are omitted here, only those parts are included, which are directly needed to provide a context for the described actions. The rest of the system can be implemented according to Figures 5.2, 5.3 and 5.4. For the full implementation of the calculator module and its interfaces, see Listings C.2 and C.3.

The applied data structures: for the implementation we use the currently supported data structures, and also consider their current limitations. The inputs are stored in three

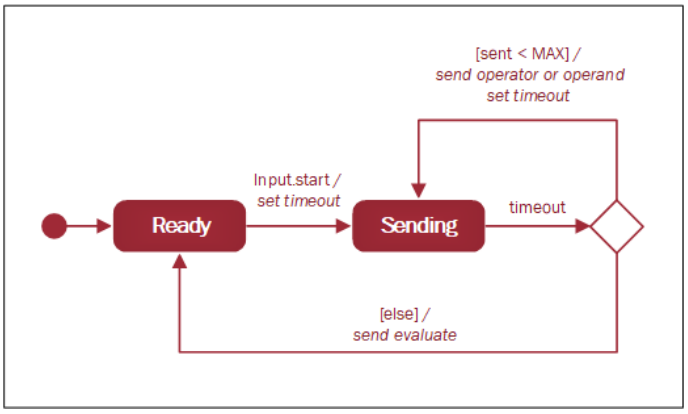


Figure 5.4: Statechart of the tester

separate arrays: one for the operators (*operators*), one for the operands (*operands*), and one for storing the sequence of operators and operands using boolean values (*isOperator*). To simulate a stack-like functionality, we also declare three integer valued variables (*numOperators, numOperands and numIsOperator*) and also keep in mind the maximum sizes of the arrays (2-2-4).

Transitions of the calculator component: As described in the previous sections, when considering the actions of the system, there are basically three different types of transitions: the ones *saving the input*, the ones *resetting the calculator and signaling error* and the one *evaluating the provided expression and signaling success.*

Saving the input: these transitions behave based on the input being an operator or an operand. Generally, they save the input into the corresponding array, at the place specified by the current state of the corresponding *num* variable. They also add a boolean value to the *isOperand* array at the place specified by the *numIsOperand* variable. Then they increase each of the used *num* variables to prepare it for the next input. The code describing this functionality can be seen in Listing 5.1.

```
transition from OneNumber to Invalid when Input.operand / {
  operands[numOperands] := Input.operand::inOperand;
  isOperator[numIsOperator] := false;
  numOperands := numOperands + 1;
  numIsOperator := numIsOperator + 1;
}
```

Listing 5.1: Transition saving an input operand

Considering the language elements, this transition contains various assignment statements, assigning different values to elements of arrays and simple variables.

Resetting the calculator and signaling error: these transitions set all the *num* variables to 0 and raise the *Output.error* event. By this, the (virtual) 'stack pointers' are returned to the base of the stack, thus making the contents unreachable (as long as the stack pointers are used in the right way). The code describing this functionality can be seen in Listing 5.2.

```
transition from OneNumber to Init when Input.operator / {
  raise Output.error;
  numOperators := 0;
  numOperands := 0;
  numIsOperator := 0;
```

```
}
```

Listing 5.2: Transition resetting the system

Considering the language elements, this transition contains one (simple) event raising, and three (simple) assignment statements.

Evaluating the input: this transition implements the modified version of the RPN left-to-right algorithm [5]. The pseudocode of the original algorithm can be found in Listing C.1, to which only modifications regarding the applied data structures were introduced – i.e. the original algorithm stores the data in one array, in this example we use three arrays, as described above. In short, the algorithm requires a stack, on which the results of the individual operations are stored. It checks the input sequence iteratively, and if the element is an operand, it pushes it onto the stack. If the element is an operator, it takes two elements from the stack and calculates the result according to the operator. When the end of the input sequence is reached, the stack contains only one element, which is the result of the calculation. After the evaluation, the result of the calculation is sent along with the success signal. In the end, the calculator is reset. The code implementing this functionality can be seen on Listing 5.3.

```
transition from MaybeValid to Init when Input.evaluate [numOperators = numOperands - 1] / {
  var finalResult : integer;

  //left-to-right algorithm variables
  var resultStack : array integer[2];

  var stackTop : integer := 0;
  var operatorsTop : integer := 0;
  var operandsTop : integer := 0;
  var isOperatorValue : boolean;

  //variables in the for loop
  var temp0 : Operator;
  var temp1 : integer;
  var temp2 : integer;
  var temp : integer;

  for (i : integer in [0 .. 4]){  //for the maximum possible input sequence
    if (i < (numOperators + numOperands)) { //if there is an input at the given index
      isOperatorValue := isOperator[i];
      if (isOperatorValue) {  //if the input was an operator, evaluate
        //get the operator and the operands
        temp0 := operators[operatorsTop]; operatorsTop := operatorsTop + 1;
        temp1 := resultStack[stackTop]; stackTop := stackTop - 1;
        temp2 := resultStack[stackTop]; stackTop := stackTop - 1;

        //calculate
        if (temp0 = ::addition) {
          resultStack[stackTop] := temp1 + temp2; stackTop := stackTop - 1;
        }
        else if (temp0 = ::subtraction) {
          resultStack[stackTop] := temp1 - temp2; stackTop := stackTop - 1;
        }
        else if (temp0 = ::multiplication) {
          resultStack[stackTop] := temp1 * temp2; stackTop := stackTop - 1;
        }
        else if (temp0 = ::division) {
          resultStack[stackTop] := temp1 div temp2; stackTop := stackTop - 1;
        }
      }
      else {  //if the input was an operand, push
        temp := operands[operandsTop]; operandsTop := operandsTop + 1;
        resultStack[stackTop] := temp; stackTop := stackTop + 1;
      }
    }
    else {  //no more input, can break
```

```
      break;
    }
  }

  //get the final result and send
  finalResult := resultStack[0];
  raise Output.success(finalResult);

  //reset the calculator
  numOperators := 0;
  numOperands := 0;
  numIsOperator := 0;
}
```

Listing 5.3: Transition evaluating the input sequence

This transition demonstrates the capabilities of multiple complex language elements. It declares local variables, iterates over a range of values, breaks the iteration when the continuation does not modify the final result anymore, and carries out operations based on the values of variables.

It is easy to see, that this solution is unnecessarily complex, and further optimizations would be possible to improve not only the readability, but also the performance of the system. This is due to the currently supported language elements and model transformations of the Gamma Framework. Nevertheless, some of these – currently only theoretical – improvements are discussed in Subsection 5.2.5.

The tester component: for the tester component, we are not going to analyze the individual transitions, variables or data structures of the component, as there is no complex algorithm to implement when sending a *random* input sequence. Instead, we are going to analyze implementation of the individual actions using the introduced action language.

The send operator and send operand actions can be implemented using choice or select statements: we want the operators and operands to be randomly selected from the possible values, and these constructs offer exactly that through non-deterministic choice. A possible implentation of these actions can be seen on Listings 5.4 and 5.5.

```
{
  . . .
  choice {
    branch[true] raise Output.operator(::addition);
    branch[true] raise Output.operator(::subtraction);
    branch[true] raise Output.operator(::multiplication);
    branch[true] raise Output.operator(::division);
  }
  . . .
}
```

Listing 5.4: Sending a random operator to the calculator

```
{
  //var toSend : integer;
  . . .
  toSend := [0 .. 100].select; //selecting a number between 0 and 100
  raise Output.operand(toSend);
  . . .
}
```

Listing 5.5: Sending a random operand to the calculator

5.2.4 Evaluation of the Results

The calculator component of the designed system was implemented using the Gamma Framework. The AST was correctly constructed from the textual description of the statechart, and no elements were marked incorrect by the validation rules. However, the model transformations (to low-level statechart, then to xSTS, then to Java) yielded unexpected results.

To provide a basis of comparison, the execution of the transformation workflow for the validation models discussed in Section 5.1 on a given setup only lasted a few seconds and resulted in .java files under 1MB.

In case of the RPN Calculator, the execution of the model transformations on the same setup, with the modeled system only able to add two numbers lasted around two minutes and resulted in a 16MB .java file. For a calculator able to add or subtract two numbers, the transformations lasted around twenty minutes and resulted in a 76MB .java file. The .java files contained no errors apart from the size of an individual method exceeding the maximum 64kB size or the OutOfMemoryExceptions when trying to compile the given file.

The problem: during the model transformations, many brute-force solutions are applied. These solutions include, but are not limited to local variables not being differentiated from global variables, array access expressions being implemented using additional variable declarations and if-statements consisting of several branches, and guards of if, choice and switch statements being extracted into separate variables and assignments, to be able to support access expressions. These actions result in the state space explosion of the system. This leads to many complex transitions, which are then merged into one xSTS transition, which is in turn transformed into one Java method that is impossible to compile and execute – even though the transformation and the resulting operation of the individual elements may be correct.

5.2.5 Further Improvements of the Calculator

If we assume that all the language elements are integrated into the Gamma Framework as described in Chapter 3, we can improve the calculator in several ways:

- The input can be stored in one array of records, eliminating the need for the modification of the left-to-right algorithm, thus making the code more efficient and readable.
- A stack type can be implemented using records containing an array and an integer, the latter containing the top of the stack.
- Variables can be declared inside a for-loop, making the code easier to understand.
- Variables can be assigned complex expressions (e.g. array access) in their initializer expressions, reducing the number of statements needed.
- The input evaluation logic can be extracted into a procedure, making the code more readable.
- The operation evaluation logic can be extracted into a procedure, making the code more readable.

These modifications are currently not possible, as either the language, or the model transformations of the framework do not support them. However, the model could be further

refined using the currently supported elements too: for instance, input buffer overflow is not handled by the given code.

Chapter 6

Conclusion

The Gamma Statechart Composition Framework is a modeling tool that supports the design, verification and code generation for reactive embedded systems. It applies model-driven software development concepts to the modeled state-based systems by supporting the use of hidden formal methods – high-level language elements designed for formal verification – and also Java code generation.

The main contribution of this work is the design and integration of an action language into the Gamma Framework, that can describe the behavior typically required of embedded systems, while also enabling the application of formal verification techniques for the language elements. This can be achieved by remaining within the boundaries of the finite-state machine formalism (and intentionally avoiding Turing completeness of the language).

The language supports different types of value declarations (variable, constant, parameter), expressions, state modification actions (variable assignments, event raisings, timeouts), control flow statements (if-else, switch-case, for-then, choice, assertion) and procedures. Its type system contains the ordinary primitive types such as integers and booleans, but also enumerations, bounded integers, arrays, records and integer ranges.

The language definition precisely describes the concrete syntax using the EBNF notation in accordance with the metamodel of the language, well-formedness constraints in the form of validation rules, and also denotational semantics for the transformation of the models to various other formalisms. The model transformations are executed in multiple steps: first of all, a low-level model is generated, which reduces the set of the applied metamodel elements and eliminates most of their redundancy. Next, the low-level model is transformed to the xSTS formalism, which only contains a handful of language elements. This xSTS model can be formally verified, and it is also suitable for the transformation to Java code, which task is delegated to the Gamma Framework.

The case study of the RPN calculator demonstrated the capabilities of the action language: similarly to other, general purpose imperative programming languages, it can be used to implement various algorithms for the solution of complex problems. The algorithms written in other languages are easily portable to the introduced action language, and in many cases, their capabilities are not limited by the restrictions applied to it. However, the case study also demonstrated the dangers and boundaries of the application of the language: even though it seemed to work for simple models, the brute-force solutions during the model transformations resulted in a state-space explosion so huge, that the calculator could not be executed in a basic environment.

As for future work, the most pressing task is the optimization of the model transformations. Based on previous experience, it is not impossible at all: by changing the code in one place, the generated Java files could be reduced to around one sixth of their original sizes (which still proved to be impossible to execute). Then, the not yet integrated language elements should be integrated into the Gamma Framework, along with the extensive validation of the behavior of the individual language elements.

By designing an imperative programming language that can be formally verified, but also offers high-level language elements, and integrating it into the Gamma Framework – a powerful tool used for modeling state-based systems – we hope to support software and system engineers to take advantage of the full potential of model-driven software development.

The action language is available as part of the Gamma Statechart Composition Framework publicly accessible on GitHub under the following link: https://github.com/FTSRG/gamma.
The model transformations to the low-level and xSTS models and the Java language are not yet publicized, but are available upon request.

List of Figures

List of Tables

Bibliography

[1] Iso/iec 14977:1996 information technology - syntactic metalanguage - extended bnf, 1996.

[2] Jung Ho Bae and Heung Seok Chae. Systematic approach for constructing an understandable state machine from a contract-based specification: Controlled experiments. *Softw. Syst. Model.*, 15(3):847–879, July 2016. ISSN 1619-1366. DOI: 10.1007/s10270-014-0440-2. URL http://dx.doi.org/10.1007/s10270-014-0440-2.

[3] 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.

[4] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-27755-2. DOI: 10.1007/978-3-540-27755-2_3. URL https://doi.org/10.1007/978-3-540-27755-2_3.

[5] Arthur W. Burks, Don W. Warren, and Jesse B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954. ISSN 08916837. URL http://www.jstor.org/stable/2001990.

[6] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 670–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04425-0.

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

[8] Alexandre David Gerd Behrmann and Kim Larsen. A tutorial on uppaal 4.0 - updated november 28, 2006. 11 2006.

[9] Bence Graics. Mixed-Semantics Composition of Statecharts for the Model-Driven Design of Reactive Systems. Master's thesis, Budapest University of Technology and Economics, Hungary, 2018.

[10] Object Management Group. Unified modeling language: Superstructure version 2.1.2 (07-11-02). Technical report, Object Management Group, 2007.

[11] Object Management Group. Object constraint language – version 2.4. Technical report, Object Management Group, 2014.

[12] Object Management Group. Action language for foundational uml (alf) - concrete syntax for a uml action language version 1.1. Technical report, Object Management Group, 2017.

[13] Object Management Group. Semantics of a foundational subset for executable uml models (fuml) version 1.4. Technical report, Object Management Group, 2018.

[14] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987. ISSN 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. URL http://www.sciencedirect.com/science/article/pii/0167642387900359.

[15] A.G. Kleppe, J. Warmer, J.B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Object technology. Addison-Wesley, 2003. ISBN 9780321194428. URL https://books.google.hu/books?id=nC6oS5xQGukC.

[16] Object Management Group. Omg unified modeling language – version 2.5.1. https://www.omg.org/spec/UML/2.5.1, December 2017. URL https://www.omg.org/spec/UML/2.5.1.

[17] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967. ISBN 0-13-165563-9.

[18] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 113–116, 2018. DOI: 10.1145/3183440.3183489.

[19] 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. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.

[20] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937. ISSN 0024-6115. DOI: 10.1112/plms/s2-42.1.230. URL https://doi.org/10.1112/plms/s2-42.1.230.

[21] 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, Jul 2016. ISSN 1619-1374. DOI: 10.1007/s10270-016-0530-4. URL https://doi.org/10.1007/s10270-016-0530-4.

Appendix A

Action Language

A.1 Overview of the Action Language

```
Action = Statement | Block ;

Block = '{'
          {Action}
        '}'
        ;

Statement = VariableDeclarationStatement |
            ConstantDeclarationStatement |
            ExpressionStatement |
            AssignmentStatement |
            RaiseEventAction |
            SetTimeoutDeclaration |
            DeactivateTimeoutDeclaration |
            IfStatement |
            ChoiceStatement |
            ReturnStatement |
            BreakStatement |
            SwitchStatement |
            ForStatement |
            AssertionStatement
            ;
```

Listing A.1: Overivew of the syntax of the action language

Appendix B

Model Transformations

B.1 Overview of the High-level-to-Low-level Transformation

High-level	**Low-level**
Variable Declaration Statement	Variable Declaration Statement(s)
Constant Declaration Statement	Variable Declaration Statement(s)
Parameter Declaration	Variable Declaration and Assignment Statement(s)
Expression Statement	Empty Action or Variable Declaration, Assignment, possibly If and Choice Statements
Reference Expression	(simple) Reference Expression or Variable Declaration, Assignment and If Statement(s)
Block	Block
If Statement	Variable Declarations, Assignments and an If Statement
Choice Statement	Variable Declarations, Assignments and a Choice Statement
Break Statement	–
Return Statement	Assignment Statement or –
Switch Statement	Variable Declarations, Assignments and an If Statement
For Statement	Variable Declarations, Assignments (possibly If or Choice Statements)
Set Timeout Action	Assignment Statement
Deactivate Timeout Action	Not yet transformed
Raise Event Action	Assignment Statement(s)
Empty Statement	Empty Statement

Table B.1: Overview of the high-level-to-low-level action transformation

B.2 Overview of the Low-level-to-xSTS Transformation

High-level Element	**xSTS Element(s)**
Variable Declaration Statement	Empty Statement, temporary Variable Declaration
Reference Expression	Reference Expression
Variable Initializing Expression	Variable Initializing Expression
Assignment Statement	Assignment Action
Block	Sequential Action
If Statement	Non-deterministic Action with Sequential Action(s) and Assume Action(s)
Choice Statement	Non-deterministic Action and Sequential Action(s) and Assume Actions(s)
Empty Statement	Empty Statement

Table B.2: Overview of the low-level statechart-to-xSTS transformation

Appendix C

Validation and Case Study

C.1 Testing the Language Elements

Test Suite	Test Case	Result
Raise Events	Test one simple event	Passed
	Test two simple events	Passed
	Test one parametric event	Passed
Variable Declarations	test one integer variable (with initialization)	Passed
	Test one boolean variable (with initialization)	Passed
	Test one integer array variable (with initialization)	Passed
	Test one boolean variable (with initialization)	Passed
	Test one record variable (with initialization)	Passed
	Test one enum variable (with initialization)	Passed
Constant Declarations	Same as the cases of variable declarations	Not tested
Expression Statements	Either at the procedures or cannot be tested	Not tested
Assignment Statements	Test *integer reference := integer literal*	Passed
	Test *integer reference := integer reference*	Passed
	Test *integer reference := integer function*	Failed
	Test *integer reference := record-integer*	Passed
	Test *integer reference := array-integer*	Passed
	Test *array-integer := integer reference*	Passed
	Test *record-integer := integer reference*	Passed
If Statements	Test *if(true)*	Passed
	Test *if(false)*	Passed
	Test *if(true reference)*	Passed
	Test *if(false reference)*	Passed
	Test *if(true)else*	Passed
	Test *if(false)else*	Passed
	Test *if(true)elsif(true)*	Passed
	Test *if(false)elsif(false)*	Passed
	Test *if(false)elsif(true)*	Not tested
	Test *if(true)if(true)*	Passed
	Test *if(true)if(true)*	Passed
Choice Statements	Test branch(true)	Passed
	Test branch(false)	Failed
	Test branch(true)branch(true)	Passed
	Test branch(false)branch(true)	Passed
Switch Statements	Test case(true)	Passed
	Test case(false)	Passed
	Test case(true)case(false)	Passed
	Test case(false)case(true)	Passed
	Test case(true)breakCase(false)	Passed
For Statements	Test for array reference, only for{}	Passed
	Test for array reference, for{break}then{}	Passed
	Test for array reference, for{noBreak}then{}	Passed
	Test for array reference, accessing the parameter values	Passed

Table C.1: Results of the validation of the individual language elements

C.2 RPN Calculator

```
for each token in the postfix expression:
  if token is an operator:
    operand_2 ← pop from the stack
    operand_1 ← pop from the stack
    result ← evaluate token with operand_1 and operand_2
    push result back onto the stack
  else if token is an operand:
    push token onto the stack
result ← pop from the stack
```

Listing C.1: The original RPN left-to-right algirithm

```
package Interfaces

interface CalcInput {
  out event operator (inOperator : Operator)
  out event operand (inOperand : integer)
  out event evaluate
}

interface CalcOutput {
  out event success (result : integer)
  out event error
}

type Operator : enum {
  addition, subtraction, multiplication, division
}
```

Listing C.2: Interfaces of the RPN calculator

```
package Calculator

import "Interfaces"

statechart CalculatorStatechart[
    port Input : requires CalcInput
    port Output : provides CalcOutput
  ]{
  // Variables
  var operators : array Operator[2]
  var numOperators : integer := 0
  var operands : array integer[2]
  var numOperands : integer := 0
  var isOperator : array boolean[4]
  var numIsOperator : integer := 0  //expression length


  // Transitions
  transition from calcEntry to Init

  transition from Init to OneNumber when Input.operand / {
    operands[numOperands] := Input.operand::inOperand;
    isOperator[numIsOperator] := false;
    numOperands := numOperands + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from Init to Init when Input.operator / {
    raise Output.error;
  }
  transition from Init to Init when Input.evaluate / {
    raise Output.error;
  }

  transition from OneNumber to Invalid when Input.operand / {
    operands[numOperands] := Input.operand::inOperand;
    isOperator[numIsOperator] := false;
```

```
    numOperands := numOperands + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from OneNumber to Init when Input.operator / {
    raise Output.error;
    numOperators := 0;
    numOperands := 0;
    numIsOperator := 0;
  }
  transition from OneNumber to Init when Input.evaluate / {
    var oneNumberResult : integer;
    oneNumberResult := operands[0];
    raise Output.success(oneNumberResult);
    numOperands := 0;
    numIsOperator := 0;
  }

  transition from Invalid to Invalid when Input.operand / {
    operands[numOperands] := Input.operand::inOperand;
    isOperator[numIsOperator] := false;
    numOperands := numOperands + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from Invalid to MaybeValid when Input.operator / {
    operators[numOperators] := Input.operator::inOperator;
    isOperator[numIsOperator] := true;
    numOperators := numOperators + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from Invalid to Init when Input.evaluate / {
    raise Output.error;
    numOperators := 0;
    numOperands := 0;
    numIsOperator := 0;
  }

  transition from MaybeValid to Invalid when Input.operand / {
    operands[numOperands] := Input.operand::inOperand;
    isOperator[numIsOperator] := false;
    numOperands := numOperands + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from MaybeValid to MaybeValid when Input.operator / {
    operators[numOperators] := Input.operator::inOperator;
    isOperator[numIsOperator] := true;
    numOperators := numOperators + 1;
    numIsOperator := numIsOperator + 1;
  }
  transition from MaybeValid to Init when Input.evaluate [numOperators = numOperands - 1] / {
    var finalResult : integer;

    // left-to-right algorithm
    var resultStack : array integer[2];

    var stackTop : integer := 0;
    var operatorsTop : integer := 0;
    var operandsTop : integer := 0;
    var isOperatorValue : boolean;

    //variables in the for loop
    var temp0 : Operator;
    var temp1 : integer;
    var temp2 : integer;
    var temp : integer;

    for (i : integer in [0 .. 4]){
      if (i < (numOperators + numOperands)) {
        isOperatorValue := isOperator[i];
        if (isOperatorValue) {
          // Get values
          temp0 := operators[operatorsTop]; operatorsTop := operatorsTop + 1;
          temp1 := resultStack[stackTop]; stackTop := stackTop - 1;
```

```
            temp2 := resultStack[stackTop]; stackTop := stackTop - 1;

            // Calculate
            if (temp0 = ::addition) {
              resultStack[stackTop] := temp1 + temp2; stackTop := stackTop - 1;
            }
            else if (temp0 = ::subtraction) {
              resultStack[stackTop] := temp1 - temp2; stackTop := stackTop - 1;
            }
            else if (temp0 = ::multiplication) {
              resultStack[stackTop] := temp1 * temp2; stackTop := stackTop - 1;
            }
            else if (temp0 = ::division) {
              resultStack[stackTop] := temp1 div temp2; stackTop := stackTop - 1;
            }

          }
          else {
            temp := operands[operandsTop]; operandsTop := operandsTop + 1;
            resultStack[stackTop] := temp; stackTop := stackTop + 1;
          }
        }
        else {
          break;
        }
      }

      //
      finalResult := resultStack[0];
      raise Output.success(finalResult);

      //
      numOperators := 0;
      numOperands := 0;
      numIsOperator := 0;
    }
    transition from MaybeValid to Init when Input.evaluate [numOperators /= numOperands - 1] / {
      raise Output.error;
      numOperators := 0;
      numOperands := 0;
      numIsOperator := 0;
    }

    // States
    region mainRegion{
      initial calcEntry
      state Init{

      }
      state OneNumber{

      }
      state Invalid{

      }
      state MaybeValid{

      }
    }
  }
```

Listing C.3: Model of the RPN calcuator