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

Formal Methods for Better Standards

Validating the UML PSSM Standard About State Machine Semantics

Scientific Students' Association Report

Author:

Péter Szkupien
Ármin Zavada

Advisor:

Márton Elekes
Bence Graics
dr. Vince Molnár

2022

Contents

Kivonat

Az emberiség egyre komplexebb biztonságkritikus rendszereket készít, amelyeket már túlnyomó többségben szoftverek vezérelnek. Ezen rendszerek hibás működése katasztrofális következményekkel járhat, ezért új fejlesztési módszertanok váltak szükségessé a biztonság garantálására. Többek között ilyen a modellalapú rendszertervezés is, amely számos előnnyel rendelkezik a hagyományos módszerekhez képest.

A modellalapú rendszertervezés használhatához elengedhetetlenek a modellezési nyelvek, amelyekkel többek között a rendszer viselkedése is leírható. A mérnöki viselkedésmodellek esetében is egyre elterjedtebb a végrehajtható modellek használata, melyek haszna abban rejlik, hogy pontos végrehajtási szemantikával rendelkeznek. Reaktív rendszerek esetében széleskörben használt modellezési nyelv a Unified Modeling Language (UML) állapotgép formalizmusa, amely működésének leírását a Precise Semantics of UML State Machines (PSSM) szabvány definiálja. Az elvárt viselkedést egy tesztkészlet is bemutatja, amely tesztmodellek lehetséges lépéssorozatait definiálja. A szabvány minősége szempontjából kiemelten fontos, hogy ez a tesztkészlet a szöveges szemantikával konzisztens legyen, és hiánytalanul mutassa be a lehetséges viselkedések körét.

Ebben a dolgozatban formálisan modellezzük az UML állapotgépek PSSM szabvány által definiált működését, valamint a tesztkészlet állapotgépeit. Az így előálló modellekből formális módszerek segítségével automatizáltan előállítjuk a lehetséges lépéssorozatokat, melyeket végül összevetünk a szabványban megadottakkal.

A bemutatott módszerrel lehetővé válik a szabvány esetleges hiányosságainak és hibáinak felfedése, amely a szabvány pontosításán keresztül segíti a jobb minőségű mérnöki modellek készítését, végső soron pedig a biztonságosabb kritikus rendszerek fejlesztését is. A folyamat egyúttal példaként is szolgálhat más modellezési nyelvek pontos tesztkészleteinek előállításához, ezáltal biztosítva a pontos működési szemantikát.

Abstract

Humanity is creating more and more complex safety-critical systems, almost all of which are now operated by software. The incorrect operation of these systems can lead to catastrophic consequences, therefore new development principles have become necessary to guarantee safety. One of these principles is model-based systems engineering (MBSE) which has several advantages over traditional ones.

To apply model-based systems engineering, modeling languages are needed for describing the structure and behavior of systems. The application of executable models is becoming more and more popular in the case of engineering models. The power of executable models lies in their precise execution semantics. In the case of reactive systems, a widely used modeling language is the state machine formalism of the Unified Modeling Language (UML), whose precise semantics is defined in the Precise Semantics of UML State Machines (PSSM) standard. PSSM also presents the expected behavior of such models with a test suite, which defines the possible execution traces of the test models. In terms of the quality of the standard, the consistency of the test suite and the semantics is especially important, as well as demonstrating the possible behaviors completely.

In this work, we formally model the behavior of UML state machines, defined by the PSSM standard, along with the state machines of the test suite. From the resulting models, we automatically generate the possible execution traces using formal methods and compare them with the ones given by the standard.

The presented technique allows the exposure of possible errors and shortcomings in the standard, which, by refining the specification, facilitates the construction of better quality models, and ultimately, the development of safer safety-critical systems as well. The workflow may also serve as an example for developing precise test cases for the modeling languages of the future.

Chapter 1

Introduction

Reactive systems are becoming ever more complex as user requirements proliferate. As a result, such systems are generally decentralized; they consist of heterogeneous components distributed among several computing nodes, which constantly interact and communicate with each other and external resources (e.g., cloud computing via the Internet) while carrying out critical tasks. This problem is amplified in safety-critical systems – such as embedded control systems in the railway, automotive, and aerospace industries – as this field requires the highest degree of confidence in the correctness of the final system [23].

During the product lifecycle, the teams of engineers go through various phases of development and create multiple design artifacts. To guarantee the consistency and correctness of the final system, engineers use verification and validation (V&V) techniques (e.g., model checking) throughout development. However, during the increased lifespan of such projects, the V&V process becomes more difficult, resulting in an increased likelihood of faults [19].

To tackle the increasing development complexity, new approaches and tools have been introduced to supervise the design, verification, and implementation of reactive safety-critical systems. *Model-based systems engineering* (MBSE) is a methodology aiming to support the development process by allowing the engineers to use high-level modeling languages with automatically derivable implementation and verifiable design [22], decreasing (but not eliminating) the need for manual V&V.

In MBSE, models are the primary artifacts of the development process [24], which are expressed using various modeling languages, that have a language structure (abstract syntax), grammatical rules (well-formedness constraints), exact graphical or lexical representation (concrete syntax) and an interpretation (semantics) of well-formed models. In order to use such models for simulation, verification, or code generation, the preciseness of the language is essential [2].

The *Unified Modeling Language* (UML) [10] is a general-purpose modeling language – developed by the *Object Management Group* (OMG) – that is widely used in MBSE approaches to describe the behavior and structure of systems. UML defines several types of diagrams that allow the user to define and visualize different aspects of the system. Since its birth in 1997, a multitude of additional specifications have been released for UML, reducing the lack of precise semantical specification of the langauge [25]. The *Semantics of a Foundational Subset for Executable UML Models* (fUML) [13] specification defined a subset of standard UML with precise structural and behavioral semantics, which has been extended in the *Precise Semantics of UML Composite Structures* (PSCS) [11] specification with composite structures. The latest development came with the release of

the *Precise Semantics of UML State Machines* (PSSM) [12] specification, which extended the executable subset with UML State Machines. Additionally, PSSM also presents the expected behavior of models using a test suite, which may be used to validate the PSSM conformance of various tools, and also helps in understanding the execution semantics.

Despite the large amount of effort that went into extending the subset of UML elements with precise semantics, there still remain un- or underspecified parts of the standard to this day [4, 3]. The reason for this lies in the immense difficulty in writing standards in a highly precise, but still understandable manner [1]. A possible way to further increase the semantical precision of UML may be to model its semantics using a formal language and generate all the execution traces for the specified test cases in a highly detailed fashion (i.e., making every step observable) using formal methods. The usage of formal methods can be automated based on existing development workflows and could produce the complete set of execution traces while reducing the possibility of remaining errors in the standard.

In this work, we propose an approach for validating modeling standards and demonstrate it using the PSSM Test Suite as a case study. We formally model a subset of the behavior of PSSM State Machines and implement several PSSM Test Models as well in the Gamma Statechart Composition Framework [20]. With the use of formal methods, we generate the test cases' execution traces using the Theta Model Checking Framework [27], by exploring the possible state space of the models. To provide highly detailed execution traces, we insert additional tracing information into the model with the modification of the model transformations. Using the resulting execution traces, we validate a subset of the PSSM Test Suite and provide a deeper understanding of the internal mechanics of PSSM State Machines.

Our main contributions are the following.

- We proposed an approach for conformance testing in modeling standards, by transforming a set of test models into *formal* test models based on the natural language specifications, and generating the execution traces.
- We formalized a subset of the underlying semantics of PSSM State Machines in the formal language of Gamma according to fUML and PSSM specifications.
- Using formal methods, we automated the generation of the complete set of execution traces from the Gamma models by systematically exploring their state space.
- We introduced a graphical way to present all the possible execution traces of a model with configurable granularity, resulting in a more intuitive representation of their differences.
- As a demonstration, we used our approach on PSSM resulting the PSSM Validation Workflow, which revealed several previously unknown errors in the Test Suite.

The rest of the work is structured as follows. Chapter 2 gives an overview of the theoretical background needed to understand our main contributions. In Chapter 3, we overview our validation workflow for modeling standards. Chapter 4 formulates the internal components of PSSM State Machines and describes the mapping from PSSM State Machines to formal Gamma models. Next, in Chapter 5, we describe how the trace generator explores the state space of Gamma models and provides the possible execution traces. Chapter 6 presents the validation workflow by evaluating a subset of the PSSM Test Suite. Lastly, in Chapter 7, we draw our conclusions on the work and present possible future work.

Chapter 2

Background

This work builds upon the theories and results of several fields across computer science, including systems engineering, modeling language semantics, formal modeling, and symbolic transition systems. Given the broad spectrum of theoretical background, this chapter introduces all the necessary preliminary knowledge this work uses as its foundation, and establishes the basis of the presented work.

The chapter is structured as follows. Section 2.1 presents the Unified Modeling Language, which is a widely used modeling language in the field of model-based systems engineering. Next, Section 2.2 introduces the Gamma Statechart Composition Framework, which enables the formal modeling of component-based reactive systems. Section 2.3 introduces the Theta Model Checking Framework used to efficiently explore the state space of formal models. Lastly, Section 2.4 showcases related works in the literature.

2.1 Unified Modeling Language

The *Unified Modeling Language* (UML) [10] is a general-purpose modeling language – developed by the *Object Management Group* (OMG) – that is widely used in the model-based systems engineering domain to describe the behavior and structure of systems. UML provides numerous types of diagrams for visualizing different aspects of systems. *State Machine Diagrams* and *Activity Diagrams* are behavioral diagrams, whose purpose is to describe *how* a component behaves in certain situations. Base UML does not specify precisely the operational semantics of the behavior models. The *Precise Semantics of UML State Machines* (PSSM) [12] is a follow-up specification for a subset of UML elements, refining their execution semantics. This section provides a brief overview of the State Machine and Activity behaviors and summarizes the PSSM standard.

2.1.1 Behavior

Behaviors are the basic concepts for modeling dynamic change in UML models [10]. A Behavior may be *executed*, either by direct invocation or through an *active object* hosting it. During execution, Behaviors may produce several types of *signals*, which they can send to themselves or to other Behaviors. The *occurrence* of a signal may *trigger* a reaction in Behaviors: signals thus provide the basic mechanism for their communication.

2.1.2 Activity Behavior

An Activity is a kind of Behavior that is specified as a graph of *nodes* interconnected by *edges* [10]. Three (disjunct) types of nodes are supported in Activities, which can be connected with two types of edges. *Executable nodes* represent lower-level steps in the overall Activity. *Object nodes* hold data that is input to and output from executable nodes and moves across *object flow* edges. *Control nodes* specify sequencing of executable nodes via *control flow* edges. Activities are essentially what are commonly called "control and data flow" models [10]. Such models of computation are inherently concurrent, as any sequencing of activity node execution is modeled explicitly by activity edges, and no ordering is mandated for any computation not explicitly sequenced. Activity execution is modeled using *token-flow semantics* – tokens flow through the network of nodes along the edges connecting them.

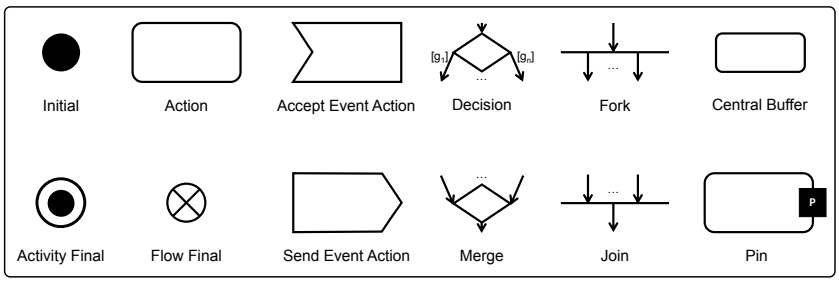


Figure 2.1: Artifacts of UML Activity Diagrams.

Figure 2.1 shows the set of modeling elements Activity Diagrams may contain. Simple *actions* represent a single step of behavior. Action nodes may have *Pin*s, which specify input or output values, and are connected by object flows. Execution starts from the *Initial* node and ends with an *Activity Final* node. *Fork* nodes *generate* tokens on all their output flows while *Join* nodes *consume* all incoming tokens. Generally, fork-join pairs model parallel behavior over multiple branches. Simple branching behavior can be modeled using *Decision* nodes, which forward the incoming token to a single outgoing flow. Their counterparts are *Merge* nodes, which merge all incoming branches into one. Activities may also receive and send *events* (a type of signal) using the *Accept Event* and *Send Signal* nodes accordingly.

Figure 2.2 presents a simple activity modeling the printing of the word `S1(entry)` to the trace. The activity first gets a reference for *this* (the execution context) and the word to be traced. The two values are forwarded to the *trace* action node with object flows, which then uses these objects. Execution terminates upon reaching the activity final node.

2.1.3 State Machine Behavior

For discrete event-driven behavioral modeling, UML defines State Machines using a finite state-machine formalism. In the following, we give a brief overview of the elements used in this work based on [10].

A *State Machine* is composed of one or more (orthogonal) *Regions*, each *Region* containing a set of *Vertices* interconnected by *Transitions*. A *Vertex* can be a *PseudoState*, *State* or *FinalState*. A *State* may be *simple* or *composite*; the latter containing other *Regions* as well: we say a state is a *substate* of a composite state containing it either directly or indirectly, while its direct container is its *parent state*. States may have *entry* and *exit* Behaviors, which are executed when the state is entered or exited, respectively. States may

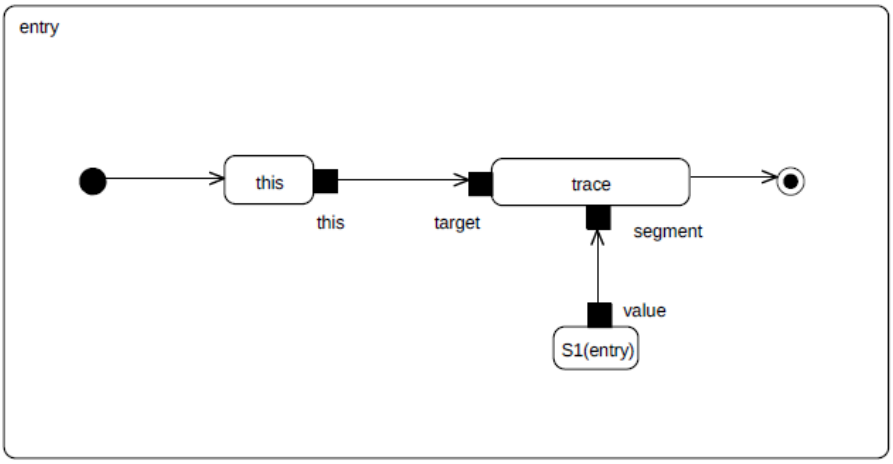


Figure 2.2: Activity Diagram of the entry behavior of S1 state in Behavior 001 [12] test case tracing the word `S1(entry)`.

also have a *doActivity* behavior, which is an Activity Behavior started *asynchronously* after the entry Behavior has completed, and runs *concurrently* to the State Machine. When the state is exited, its associated doActivity is terminated.

A *Transition* is a directed arc from a *source* Vertex to a *target* Vertex [10]. Transitions may have an *effect* Behavior, which is executed upon firing the transition. A Transition is *enabled* if its source State is the active state and it has a Trigger matching the incoming Event. Transitions may also have a *guard* constraint, which has to be evaluated to `true` to enable the transition.

More than one Transition can be enabled within a State Machine. In such cases, they may be in *conflict* with each other. Two Transitions are said to be in conflict if the intersection of the set of States (and sub-states) they exit (source state configuration) is non-empty. There are multiple ways to resolve transition conflicts. Transitions that occur in orthogonal Regions may be fired simultaneously. Otherwise, a partial ordering is defined by firing priorities. In general, let t_1 and t_2 be transitions in the State Machine, with s_1 and s_2 as their source States, respectively.

- If s_1 is a direct or indirect substate of s_2, then t_1 has higher priority than t_2.
- If s_1 and s_2 are not in the same state configuration, then there is no priority conflict between the two transitions.

In the case when there are still conflicting enabled transitions after the partial ordering, a non-deterministic choice is made between them, resulting in a *maximal* set of enabled transitions to be fired at the same time [10].

Internal transitions are special kinds of transitions, with the only difference being they do not exit or enter states – upon firing an internal transition, only its effect Behavior is executed. Internal transitions are only in conflict with transitions that exit their state.

Completion transitions are also special kinds of transitions, which are triggered by a *completion event* of its source state (*CE(source)*). A completion event is generated for a state when all of its internal activities are completed (entry and doActivity) and all of its

internal Regions have reached a FinalState. Completion events have priority over regular events.

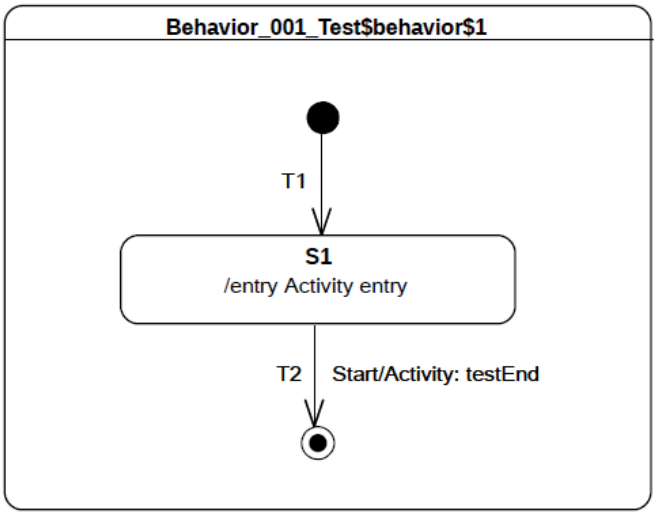


Figure 2.3: Target State Machine Diagram of the Behavior 001 [12] test case.

Figure 2.3 shows an example State Machine Diagram with a single State and two pseudostates – an initial and a final. Execution begins by entering state *S1* and firing its *entry* action. Afterwards, as soon as the *Start* event is received by the State Machine, the *T2* transition is fired – executing the *testEnd* effect, and terminating execution.

2.1.4 Precise Execution Semantics of fUML and PSSM

The *Precise Semantics of UML State Machines* (PSSM) [12] standard is the logical continuation of previous specifications created by OMG (fUML [13], PSCS [11]). It is intended to further detail the execution semantics of UML State Machines, by refining the specification with a more detailed execution model specified in natural language. In the following, we give a brief overview on the specification of fUML and PSSM based on [13, 12].

In fUML, the execution of a model begins with a `Locus`, which is an abstraction over physical or virtual computers [13]. During execution, an `Executor` is created, which can evaluate value specifications and execute behaviors synchronously or start them asynchronously. The execution semantics is defined using visitor classes for specific behaviors. For example, activity execution is defined using an `ActivityExecution` class, whose `execute()` function defines how and when the activity's nodes should be activated [13].

All objects have a common behavior defined in the `ObjectActivation` class. Active objects may communicate with each other using signals, either synchronously or asynchronously. Active objects have an *event pool*, to which incoming events are put, and from which events are dispatched to `EventAccepter` instances of the object. Event accepters can be registered at any time during the execution of behavior objects [13].

PSSM defines classes that extend the execution model of fUML [12]. A state machine is represented by an `SM_Object` class, and its semantics is defined in the `SM_ObjectActivation` class. A single, special event accepter is used for state machines, which is responsible for collecting matching transitions, calculating priorities, resolving conflicts, and selecting the set of transitions to fire in the case of orthogonal regions.

When the State Machine receives an event, its event accepter accepts it, and triggers a *run-to-completion* (RTC) in the State Machine, processing the event and taking the State Machine into the next state configuration.

DoActivities have special `DoActivityContextObjects` which provide a specialized context object for the activity. DoActivities register event accepters in the state machines activation class upon executing *AcceptEventActions*, in order to be able to accept incoming events, and trigger RTC steps in the doActivity.

An RTC step in an executing doActivity Behavior is triggered by the acceptance of an `EventOccurrence` dispatched from the `StateMachineExecution` context Object's `ObjectActivations`'s `eventPool`. This starts a new RTC step for the doActivity Execution, asynchronously from the `StateMachineExecution`. An RTC step entails the passing of tokens downstream in the activity until there is no more node capable of accepting a token [12]. During an RTC step the Activity may not accept incoming signals, however, it can be terminated.

Note that, in general, an executing doActivity Behavior will compete with the executing State Machine that invoked it to accept event occurrences dispatched from the same event pool. An activity completion occurs when, after an RTC step, there are no more event accepters registered for the doActivity [12]. When a doActivity execution completes, the StateActivation that invoked that doActivity may have to complete too. In this situation, upon the completion of the doActivity execution, a `CompletionEventOccurrence` is generated for the StateActivation and placed in State Machine's event pool.

2.1.5 Overview of the PSSM Test Suite

The PSSM test suite provides 103 test cases grouped into 18 different packages based on which part of the semantics they test. The tests were manually created by experts based on 113 requirements extracted from the UML specification.

Each test case consists of a *Target* State Machine, a *Tester* component, and a *Semantic Test* component orchestrating the case itself. Figure 2.4 illustrates the different components and their interactions. The main orchestrator of the tests is the *Semantic Test Suite* component, which instantiates and starts each *Semantic Test* by sending them a *Start* signal one by one. Upon receiving the *Start* signal, the *Semantic Test* component instantiates the *Tester* and *Target* components, connects them, and sends each a *Start* signal. During the testing phase, the *Tester* component sends the *Target* component predefined *Messages*. These messages can be any of the following: `Continue`, `AnotherSignal`, `Pending`, `Data(value : boolean)` or `IntegerData(value : integer)`, with the last two signals having value *arguments* as well. While processing the aforementioned signals, the *Target* component records its *execution trace* by calling a special `trace()` function with the desired string. At the end of the test, the *Target* component sends a *testEnd* event to the *Semantic Test*, which includes the trace list generated during execution. By comparing this trace to a predefined set of valid traces, the *Semantic Test* component can decide whether this test has *passed* or *failed*. The *Semantic Test Suite* then aggregates the results, and provides a final verdict.

Let us take the Behavior 001 test case for example. Figure 2.3 shows the behavior of the Target component. This test case specifies *empty behavior* to the Tester component (i.e., no signals sent to the Target component), so it is not shown. Each test case is provided with a description, a set of valid traces, and an example trace with RTC steps. For this model, PSSM provides the following test trace.

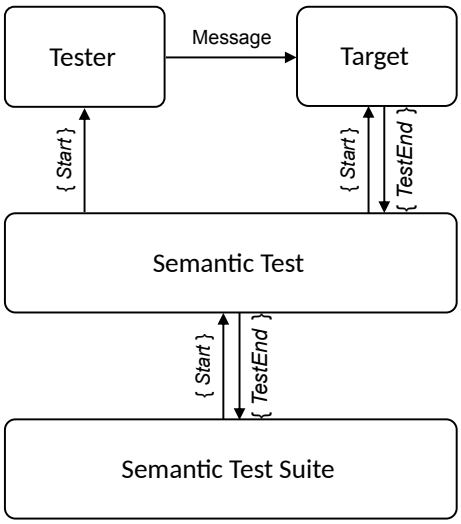


Figure 2.4: Abstract structure of a test case from the PSSM Test Suite.

- `S1(entry)`

Table 2.1 shows the RTC steps provided for this execution of the model. The different rows describe the state of the system at various steps during the RTC process. First, it fires *T1* and enters *S1* as part of the *initial RTC step*. After the initial RTC step, a *completion event* is created for state S1 after its *entry behavior* has finished execution. Since there is no *completion transition* from S1, this event is discarded. Given, that the component already received the *Start* event[1], it immediately accepts it and triggers T2, thus a *testEnd* signal is sent with the trace `S1(entry)` to the Semantic Test component. According to the specification, this is the only valid trace for this model.

Step	**Event pool**	**State machine configuration**	**Fired transitions(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[S1]	[]
3	[**Start**]	[S1]	[T2]

Table 2.1: The run-to-completion steps for execution of the Behavior 001 [12] test case.

2.2 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework[2] [20] is an integrated tool to support the design, verification, and validation of, as well as code generation for component-based reactive systems. The behavior of each atomic component is captured by a statechart, while assembling the system from components is driven by a composition language. Gamma supports several compositional semantics which can be combined, allowing the user to model systems with various execution and interaction semantics.

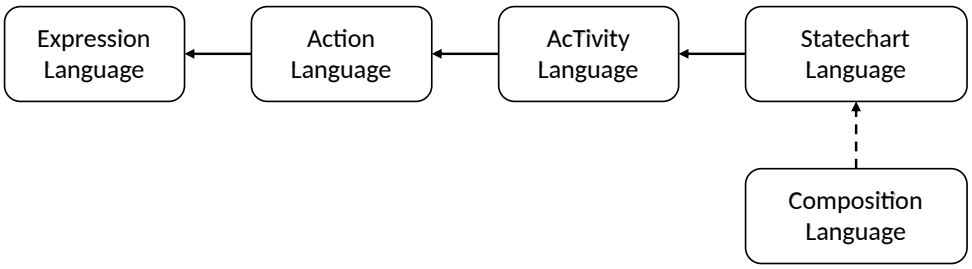


Figure 2.5: The behavioral-language structure of the Gamma Framework.

2.2.1 Gamma Behavioral Languages

Figure 2.5 displays the language structure of the behavior languages. Solid lines represent inheritance between the languages, while dashed lines represent usage dependency.

For the purposes of modeling various system *behaviors*, Gamma defines several formal languages, of which the *Gamma Expression Language (GEL)* and *Gamma Action Language (GAL)* [28] serve as the foundation. GEL and GAL together define *variables*, *types* and *expressions* accessing and combining them using *arithmetical* and *logical* expressions. GAL builds on these by providing simple *atomic actions* over variables in a reusable fashion.

In our previous work [30], we proposed the *Gamma AcTivity Language* (GATL) which is an extension of the Gamma Action Language, providing control- and data-flow semantics for modeling concurrent systems. GATL provides *simple* and *composite* actions, *fork-join* and *decision-merge* control nodes, and *action pins* for data flow modeling. Its behavior closely resembles UML Activity Behaviors [30].

The most basic building blocks of Gamma components are atomic components, of which Gamma currently supports statecharts with the *Gamma Statechart Language* (GSL) [5]. Statechart formal semantics closely resembles UML State Machine semantics, and provides simple and composite states, entry-exit and doActivity behaviors, orthogonal regions, and transitions with effects.

Listing 2.1 shows a simple Gamma statechart with an internal doActivity counting incoming events. The statechart has a port called *incoming*, through which the component may receive events. It has two states: the initial state is *Counting*, which has an associated doActivity called *CountActivity*, while *NotCounting* has no associated behavior. Upon receiving an incoming *stop* event, the statechart transitions from *Counting* to *NotCounting* – which terminates the execution of the *CountActivity*. *CountActivity* has a simple loop, in which it firsts *waits* for any event from port *incoming*. Upon receiving an event, it continues execution with the *Increment* node – which increments the value of the *count* variable.

2.2.2 Gamma Composition Semantics

As shown in Figure 2.5, the *Gamma Composition Language* uses the Gamma Statechart Language as its main behavioral language. Indeed, atomic component behavior can be modeled using statecharts; however, Gamma also provides a powerful composition language to combine different kinds of components to model various interaction and execution

[1] This is not true generally, as there is no time constraint for the incoming events.
[2] `https://inf.mit.bme.hu/en/gamma`

```
statechart CountingStatechart [
    port incoming : provides Events // exposed port of the statechart component
] {
    var count : integer := 0
    region Main {
        initial MainEntry
        state Counting {
            do / call CountActivity; // doActivity
        }
        state NotCounting
    }
    transition from MainEntry to Counting // initial state transition
    transition from Counting to NotCounting when incoming.stop // transition with a trigger
    activity CountActivity {
        initial Init // initial node
        merge Merge
        trigger Waiting when incoming.any // AcceptEventAction
        action Increment : activity [language=action] { // action node incrementing the variable
            count := count + 1;
        }
        // Specifying control flow edges between the nodes
        control flow from Init to Merge
        control flow from Merge to Waiting
        control flow from Waiting to Increment
        control flow from Increment to Merge
    }
}
```

Listing 2.1: Gamma statechart which counts incoming events using a doActivity.

semantics. In this section, we provide a detailed overview of the composition semantics based on [6].

Components serve as types of component instances. They may be *atomic* or *composite*, *synchronous* or *asynchronous*. A component can have zero or more ports, which serve as the only point of interaction between components. This ensures that external dependencies and interactions are explicitly modeled, leading to fully encapsulated behavior.

Atomic components can be considered black boxes, with

- a set of states with a well-defined initial state,
- a set of input and output events,
- a transition function that constructs the components' new state and output events from the current state and incoming events.

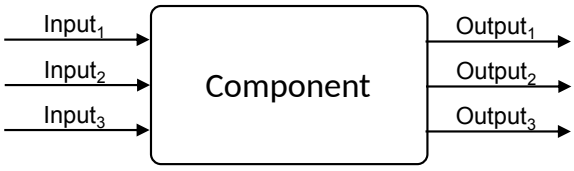


Figure 2.6: Abstract diagram of atomic components.

Synchronous Components

The execution of synchronous components is scheduled by a scheduler, which invokes the component using the *cycle*[3] input. The execution of atomic components follows a turn-based semantic, where a turn is called a *cycle*. In a cycle, the component processes its incoming signals and produces output signals in accordance with their internal states. Output signals are present for a single execution cycle only, meaning the signal disappears after one cycle. An illustration of an abstract atomic component is shown in Figure 2.6, depicting a component with a set of input and output signals.

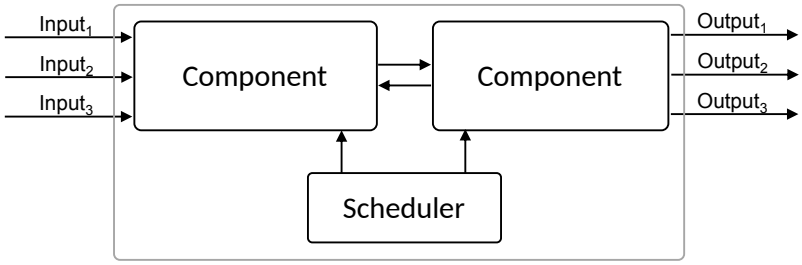


Figure 2.7: Structure of a synchronous composite component.

Synchronous *composite* components are defined by their internal components and their connections. Composite components may contain one or more *channels*, which connect internal components together. The composite component's ports may also be *bound* to an internal component's port, exposing it to the environment. The behavior of the component is defined by the scheduler, which executes the internal components. This execution may be in any custom order. Figure 2.7 depicts a generic composite component and its internal structure.

Asynchronous Components

Asynchronous behavior may be supported by injecting buffers between the components. In Gamma, *event queues* can be used to achieve the delayed transmission of events from one component to another. Event queues may contain multiple events and have priorities over each other, which in turn can affect *when* a component is scheduled. In order to use atomic components in asynchronous contexts, an *asynchronous adapter* must be used, which wraps the component, making it compatible with asynchronous systems. Figure 2.8 depicts an asynchronous adapter component with two separate event queues. $Queue_{1\text{-}2}$ has a priority of 1, while $Queue_3$ has a priority of 2 – thus events from $Input_1$ and $Input_2$ will be processed before events from $Input_3$.

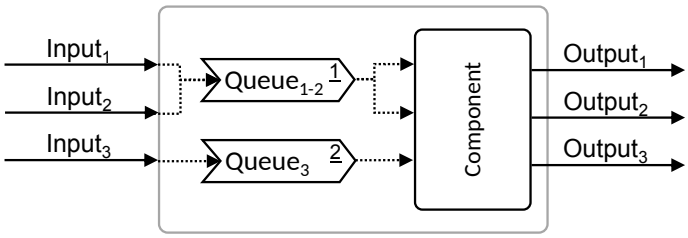


Figure 2.8: Diagram of an asynchronous adapter component.

[3]Cycle is a special implicit input of all components.

Asynchronous composite components compose other asynchronous components, connecting them with channels – similarly to synchronous components. Figure 2.9 depicts an asynchronous composite component with two internal components and queues. Asynchronous components are inherently nondeterministic, meaning there is no guarantee on the execution time and frequency of the components, only on the ordering between the processing events – events in higher priority queues will be processed first, in the order of their arrival.

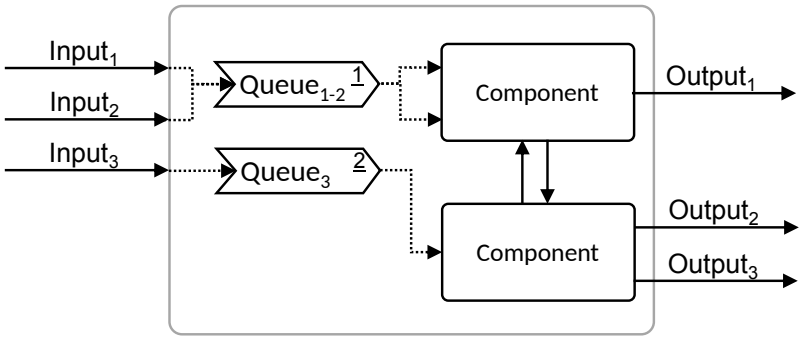


Figure 2.9: Structure of an asynchronous composite component.

```
1  adapter AsyncCounter of component counter : CountingStatechart {
2      when any / run // any incoming message will execute the wrapped component
3
4      queue incomingQueue(priority = 1, capacity = 10) {
5          incoming.any
6      }
7  }
8
9  async System {
10     component countingStatechart : AsyncCounter // instantiating the async adapter
11     component eventSupplier : EventSupplier // instantiating another async component
12
13     // connecting the components using a channel
14     channel [ eventSupplier.events ] -o)- [ countingStatechart.incoming ]
15 }
```

Listing 2.2: Gamma system of an EventSupplier component and the CountingStatechart introduced before.

Listing 2.2 depicts a simple asynchronous composite component in Gamma, connecting the previously created *CountingStatechart* to an *EventSupplier* component[4]. Since statecharts are synchronous components, we need to create an async adapter, called *AsyncCounter*. It has a single event queue with a capacity of 10. The two components are *instantiated* in the async *System*, and are connected using a *channel* between them.

2.3 Theta Model Checking Framework

The Theta Model Checking Framework[5] [27] is a generic, modular, and configurable model-checking framework. It can handle multiple formalisms as input, such as timed automata, control flow automata, and transition systems. In this work, we use the Extended Symbolic Transition System (XSTS) [21] formalism as the input of Theta.

[4]The implementation of this component is omitted for brevity.

[5]`https://inf.mit.bme.hu/en/theta`

2.3.1 Extended Symbolic Transition System (XSTS)

Extended Symbolic Transition System (XSTS) [21] is a suitable low-level formalism to describe higher-level reactive systems, such as state machines. Informally, an XSTS model describes variables and transition sets, and in every step a non-deterministically selected atomic transition fires from the appropriate transition set. In this work, we introduce XSTS based on our previous work [26].

Definition 1 (Extended Symbolic Transition System). Formally, we define an XSTS model as a 4-tuple $XSTS = \langle V, Tr, In, En \rangle$ where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of *variables* with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$, e.g. *integer*, *bool* ($\top$ for *true*, $\bot$ for $false$), or *enum*. An *enum* domain is just syntax sugar, a set of *literals* which are different values with a textual representation.
- A state of the system is $s \in S \subseteq D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$, which can be regarded as a value assignment: $s(v) \in D_v$ for every variable $v \in V$.
- $Tr \subseteq S \times S$ is the *internal transition relation*, describing the behaviour of the system itself;
- $In \subseteq S \times S$ is the *initial transition relation*, describing the initialization of the system, which is executed only once at the beginning of the execution;
- $En \subseteq S \times S$ is the *environmental transition relation*, describing the environment which the system is interacting with;
- Both Tr, In, and En may be defined as a union of exclusive transitions that the system can take. Abusing the notation, we will denote these transitions as $t \in Tr$ which actually means that $t \subseteq S \times S$ as a transition relation is a subset of Tr. ▪

A *concrete state* of the system is $c \in C = D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$, which is a value assignment $c : v \mapsto c(v) \in D_v$ for every variable $v \in V$. A concrete state c can also be described with a logical formula $\varphi = (v_1 = c(v_1) \wedge \cdots \wedge v_n = c(v_n))$ where $var(\varphi) = V$.

An *abstract state* of the system $s \in S = 2^C$ may cover more concrete states. Instead of assigning a concrete value, it assigns a set of possible values $s(v)_i$ to every variable $v \in V$: $s : v \mapsto s(v) \subseteq D_v$. If there is no restriction on variable v, i.e. it can have any value from D_v, we use the notation $s(v) = \top \triangleq s(v) = D_v$. The logical formula of an abstract state is $\varphi = ((v_1 = s(v_1)_1 \vee \cdots \vee v_1 = s(v_1)_{|s(v_1)|}) \wedge \cdots \wedge (v_n = s(v_n)_1 \vee \cdots \vee v_n = s(v_n)_{|s(v_n)|}))$ where $var(\varphi) = V$, and it covers exactly $|s(v_1)| * \cdots * |s(v_n)|$ (maybe infinite) concrete states. We use the notation $0 \leq |s| \leq 2^{|C|}$ for the number of concrete states covered by abstract state s.

Note, that every concrete state c is also an abstract state s covering only 1 concrete state c, so $|c| = |s| = 1$. Thus, without the loss of generality, we use abstract states in the following even without explicitly stating that a state is abstract.

Each transition relation $T \in \{Tr, In, En\}$ is a set of transitions t where a transition leads the system from a state s to a successor states s': $T \subseteq \{t = (s, s') \in S \times S\}$.

Every domain D has an initial value $IV(D) \in D$ e.g., $IV(bool) = \bot$, $IV(integer) = 0$. Every variable v can have a custom initial value $IV(v) \in D_v$ but it is not necessary, because its domain D_v always has one. The *initial state* s_0 is given as the *initial value* for each variable v: $s_0(v) = IV(v)$ if $IV(v)$ exists, otherwise $s_0(v) = IV(D_v)$. The execution of the system starts with assigning the initial value $s_0(v)$ to every variable $v \in V$.

From the initial state s_0, In is executed exactly once. Then, En and Tr are executed in alternation. In state s, the execution of a transition relation T (being either of the transition relations) means the execution of exactly one non-deterministically selected $t \in T$ transition. Transition t is enabled if $t(s) \neq \emptyset$. If a transition is not enabled, it can not be executed. If $\forall t \in T : t(s) = \emptyset$, transition relation T can not be executed in state s. In addition to the non-deterministic selection, transitions may be non-deterministic internally, therefore even in the case of a concrete state c, $t(c) = \{c'_1, \ldots, c'_k\}$ yields a set of successor concrete states. In other words, in the case of a general transition $t = (s, s')$, there is no restriction on the relation between $|s|$ and $|s'|$.

XSTS Operations

Transitions are described as $op \in Ops$ *operations*, which may be atomic or composite operations. The semantics of transitions are defined through the semantics of operations, which is, in turn, the definition of op as a relation over $S \times S$. For a precise description, refer to [21] – for this work, an informal definition is sufficient.

XSTS defines the following *basic operations* which lead the system from state s to successor state s':

- *Assignments*: An assignment of form $v := \varphi$ with $v \in V$ and φ as an expression of the same type D_v means that φ is assigned to v in the successor state s' and all other variables keep their value. Formally, $s'(v) = \varphi \wedge s'(v') = s(v')$ for every $v' \neq v \in V$, while $|s'| = \frac{|s|}{|s(v)|}$.
- *Assumptions*: An assumption of form $[\psi]$ with ψ as a Boolean expression over the variables ($var(\psi) \subseteq V$) checks condition ψ without modifying any variable and can only be executed if ψ evaluates to *true* over the current state s, in which case the successor state is $s' = s$, and $|s'| = |s|$ – otherwise the set of successor states is the empty set $\emptyset$, and $|s'| = 0$.
- *Havocs*: A havoc of form $havoc(v)$ with $v \in V$ means a non-deterministic assignment to variable v, i.e., after execution, the value of v can be anything from D_v and all other variables keep their value. Formally, $s'(v) = \top \wedge s'(v') = s(v')$ for every $v' \neq v \in V$. Therefore, c'_i will be $|s'| = |D_v| * |s|$.
- *Local variables*: A local variable can be declared as an operation of form $var\ v_{loc} : type := \varphi$.[6] A local variable can only be accessed in its *scope* which is its direct container composite operation. Technically, the declaration of a local variable v_{loc} adds it to V and assigns its initial value φ to v_{loc} while the end of every scope removes every local variable declared in it from V. Thus, local variables increase the state space *only* inside their container transitions. Due to the atomicity of transitions, local variables do not modify the state space of the system itself. Formally, $V' = V \cup \{v_{loc}\}$, $s'(v_{loc}) = \varphi$, $s'(v) = s(v)$ for every $v \in V$, and $|s'| = |s|$.

Composite operations contain other operations but their execution is still atomic. Practically, this means that the contained operations are defined over transient states and the composite operation determines which one(s) will be the (stable) result of the composite operation. XSTS defines the following composite operations:

[6] The default value of the type is used as an initializer unless explicitly specified by the modeler.

- *Sequences*: A sequence of form $op_1, \ldots, op_n$ is composed of operations $op_1, \ldots, op_n$ with $op_i \in Ops$ executed sequentially, each applied on every successor state of the previous one (if any). The successor state after executing the sequence is the result of the last operation. Each operation $op_{i+1} = (s_{i+1}, s'_{i+1}) = (s'_i, s'_{i+1})$ works on the result of $op_i = (s_i, s'_i)$, so $s'_i = s_{i+1}$. Thus, the transition of the sequence itself is (s_1, s'_n) but it can be executed only if $s'_i \neq \emptyset$ for every $1 \leq i \leq n$, i.e. all assumptions are satisfied.

- *Choices*: A choice of form op_1 or ... or op_n means a non-deterministic choice between operations (branches) $op_1, \ldots, op_n$ with $op_i \in Ops$. This means that exactly one executable branch op_i will be executed. A branch $op_i = (s_i, s'_i)$ can not be executed if $s'_i = \emptyset$, i.e. an assumption does not hold in the branch. If there are both executable and non-executable branches, an executable one must be executed. If all branches are non-executable ($s'_i = \emptyset$ for every $1 \leq i \leq n$), the choice itself is also non-executable, so its successor state is $\emptyset$. Generally, the set of successor states is the union of the results of any branch $\cup_{i=0}^{n} s'_i$.

- *Conditionals*: A conditional of form (ψ) ? op_{then} : op_{else} with ψ as a Boolean expression over the variables ($var(\psi) \subseteq V$) checks condition ψ, and executes $op_{then} = (s_{then}, s'_{then})$ if ψ evaluated to true, otherwise $op_{else} = (s_{else}, s'_{else})$ (op_{else} can be empty, i.e. a 0-long sequence, when $s_{else} = s'_{else}$). The sccessor state of the conditional (s, s') is $s' = s'_{then}$ if ψ is true over the variable values of s, otherwise $s' = s'_{else}$.

- *Parallels*: A parallel of form $op_1 \,||\, \ldots \,||\, op_n$ means the parallel execution of operations (branches) $op_1, \ldots, op_n$ with $op_i \in Ops$. The parallel execution means, that one substep of the parallel execution is a substep of a non-deterministically selected branch, which has not finished its execution. The parallel action finishes when all of its branches have finished.

Note that assumptions may cause any composite operation to yield an empty set as the set of successor states. This allows us to use the *choice* operation as a guarded branching operator, ruling out branches where an assumption fails by yielding an empty set as the result of that branch.

In this work, we make the following assumptions, which can be easily guaranteed by simple pre-processing.

1. The operation of transitions and non-sequence composite actions must be composite actions. Thus, single basic operations will be treated as 1-long sequences.

2. We assume that there are no sequences directly inside sequences.

These restrictions help the clarity and consistency of local variable scopes without the loss of generality.

Transition Granularity in XSTS Models

The execution of a transition relation means a non-deterministic choice over the transitions of the relation. In addition, transitions can also be non-deterministic internally. After executing transition $t = (s, s')$ from state s, we can only observe state s' but the possible internal non-determinism of t remains invisible. To make the execution of the system

fully explainable we have to make every non-deterministic choice observable. This can be achieved by making internal non-determinism external by splitting the transitions into smaller ones.

The basics of this splitting approach are presented in our previous work [26]. In Appendix A, we extend and formalize the splitting model transformation in order to make the execution traces fully explainable without changing the original semantics of the XSTS model.

2.4 Related Work

Elekes et al. in [4] investigate the assessment of modeling language specifications in regards to (i) whether they contain errors, contradictions or ambiguities, (ii) how suitable they are for assessing the correctness of related modeling tools, and (iii) how helpful they are for professionals to understand the language. They use the PSSM [12] specification as a case study to show the typical issues in modeling language specifications. As a result, they have pinpointed several significant errors in the PSSM Test Suite specification and test traces, resulting in unclear semantics. This work serves as our main motivation, to explore the possibilities of generating consistent and complete execution traces for test models using formal methods.

Ma et al. in [18] (i) conduct an extensive literature review on existing domain-specific modeling methods (DSMM) engineering approaches, (ii) provide a detailed description of validation and verification for each phase of DSMM engineering, (iii) and a road-map encompassing the desiderata for further advances in V&V in DSSM engineering. The authors advocate for the use of formal methods in DSMM engineering, however, also acknowledge that the field of formal methods is often not part of conceptual modeling courses. Given this result, a validation workflow harnessing the power of formal methods *without* explicit need for preliminary studies on formal methods can help DSMM engineering processes. Our work provides the foundation for a fully automatic validation workflow with *hidden* formal methods.

Lima et al. in [15] present a novel refinement and analysis framework for models specified in SysML[7] [8] using the COMPASS Modeling Language (CML) [29]. Their solution includes the modeling of SysML State Machines in a state-rich process algebra with separate components for the doActivities, communicating with each other asynchronously through channels and special messages. Our work is similar in the sense that it uses special *activity components* and *statechart components* in *asynchronous* systems.

Liu et al. in [16] formalized the execution semantics of UML [10] State Machines with entry and exit behaviors, concurrent doActivities, orthogonal regions and deferred event pools in the model checker called USMMC [17]. They evaluated their solution's validity by executing formal verification on models from the literature. Their work lacks a formally defined action language, while we define doActivities using the Gamma Action Language (GAL) [28] and our previous work, the Gamma AcTivity Language (GATL) [30].

[7] SysML is a variant of UML.

Chapter 3

Validation of Modeling Language Specifications

Standards play a central role in engineering methods. In the case of modeling languages, standards guarantee consistent interpretation of models between different participants. Several methods have been proposed for specifying modeling languages, but the most used method is still largely based on extensive natural language specification [1]. This results in an increased possibility of imprecise standards, which hinders the useability of the modeling languages they specify. According to previous work [4, 3] the PSSM Test Suite has several linguistical and semantical errors. This chapter introduces a *validation workflow* that uses formal methods for validating modeling language standards, which can help improve their quality. The validation workflow is also demonstrated on the PSSM Test Suite throughout the rest of the work.

The chapter is structured as follows. Section 3.1 introduces the process of standardization development and proposes the use of the validation workflow. Section 3.2 introduces the PSSM validation workflow, which serves as a case study in this work. Finally, Section 3.3 showcases an example PSSM Test Case used throughout the rest of the work.

3.1 Validation Workflow

Traditionally, the creation of a new standard begins with a *request-for-proposal* (RFP) submitted by a central agency (e.g. the creation of SysMLv2[1] [7] began with an RFP [9] created by OMG). Figure 3.1 illustrates a generic standardization process, in which *domain experts* create a *standard proposal*, for which *feedback* is given by a team of *reviewers*. This process is repeated until the desired quality is reached, at which time the standard is *released* to the general public. Any remaining error in the standard at this point might propagate into the *systems* created by *systems engineers*, which can result in unexpected and hard-to-debug behavior in the final products [23].

The standardization and review processes are both done by a limited group of people, are difficult to execute, and take a long time overall. To support this process, we propose a validation workflow, that uses formal methods to provide validation results and execution traces for example models (see Figure 3.2a). This additional *feedback* can support the refinement of specification, ultimately resulting in safer models created by systems engineers [18].

[1] The successor of the well-known language SysML [8] by OMG.

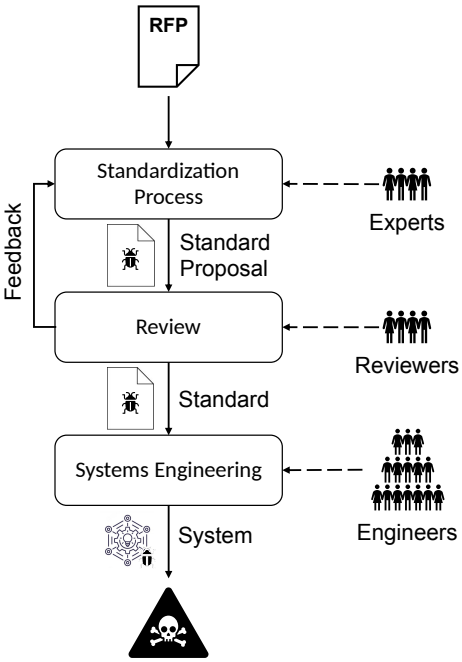
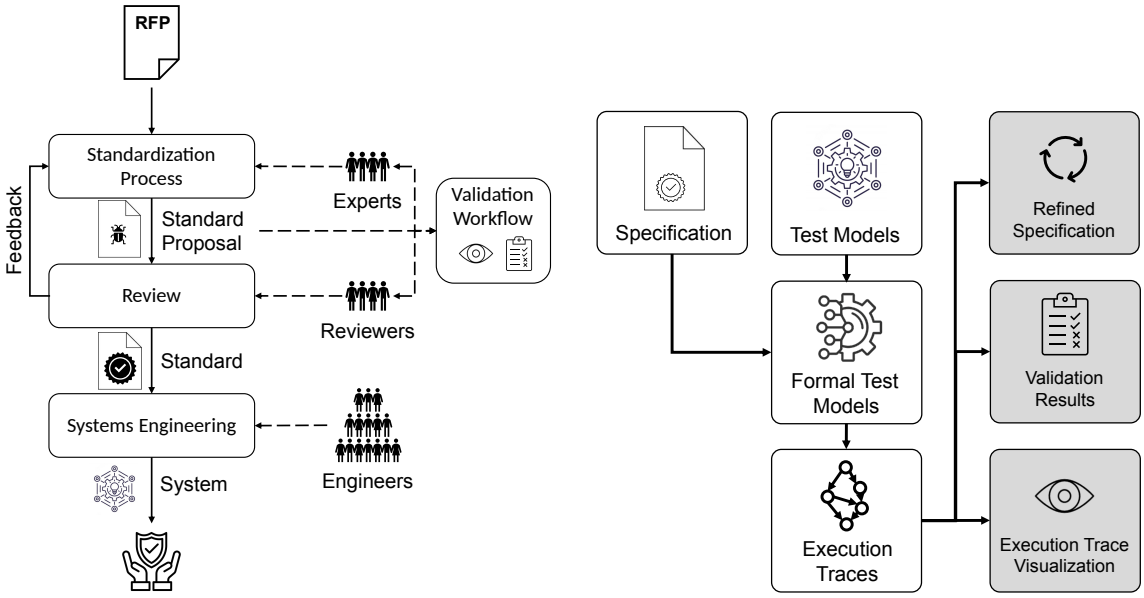


Figure 3.1: Traditional standardization process.



(a) Traditional standardization process supported with our validation workflow.

(b) High-level overview of the validation workflow.

Figure 3.2: Our proposed standardization process supported with formal methods.

In general, the validation workflow is composed of three distinct steps. The high-level overview of the workflow is shown in Figure 3.2b.

1. Based on the *semantics* provided by the standard, the *Test Models* are transformed into *formal test models*.
2. Then, with the use of formal methods, all the possible *execution traces* are generated from the formal test models.
3. Finally, the expected behavior of the models may be validated using the execution traces.

Of the steps defined above, the first two may be automated to provide *hidden* formal methods.

The validation workflow provides three main contributions (gray boxes in Figure 3.2b) which can improve standardization processes in general:

- It visualizes the possible execution traces for the test models, which
- allows the user to validate the expected behavior of the test models, in doing so
- it provides the team of experts with valuable insight for refining the specification.

3.2 PSSM Validation Workflow

To demonstrate the use of the validation workflow, we provide a case study of validating the PSSM Test Suite. As previously discussed, PSSM defines the precise *semantics* of UML State Machines in natural language specification and also showcases it using a set of test models.

Figure 3.3 depicts the validation workflow of the PSSM standard. ① The workflow starts with the *semantics specification* and the *test model*. The *gray boxes* represent artifacts that do not change from test to test, while *white boxes* are unique for each test case. ② Using the specification we construct the *Common Gamma Components* and a *Gamma Test Model* for each PSSM Test Case. The Common Gamma Components embody the common behavior of PSSM State Machines, and are reused between test cases. By combining the Gamma Test Model with the Common Gamma Components, we construct the *Gamma Composite Test Model*, which represents the test case in the Gamma language (see Chapter 4). ③ This model is then processed by Gamma using automatic *model transformations*. Firstly, it is enhanced with additional *trace calls*, which results in the *Detailed Test Model*. Afterwards, the model is transformed into the *XSTS* formalism, which is then split up, resulting in the *Split XSTS Test Model* artifact (see Chapter 5). ④ The split version of the test model is then processed by Theta, which constructs an *Abstract Reachability Graph* from the model. The *Execution Traces* are generated by exploring said graph. Finally, the execution traces are combined into one visual trace, providing the behavior of the model in a compact way, showcasing the trace calls during the possible executions (see Chapter 5).

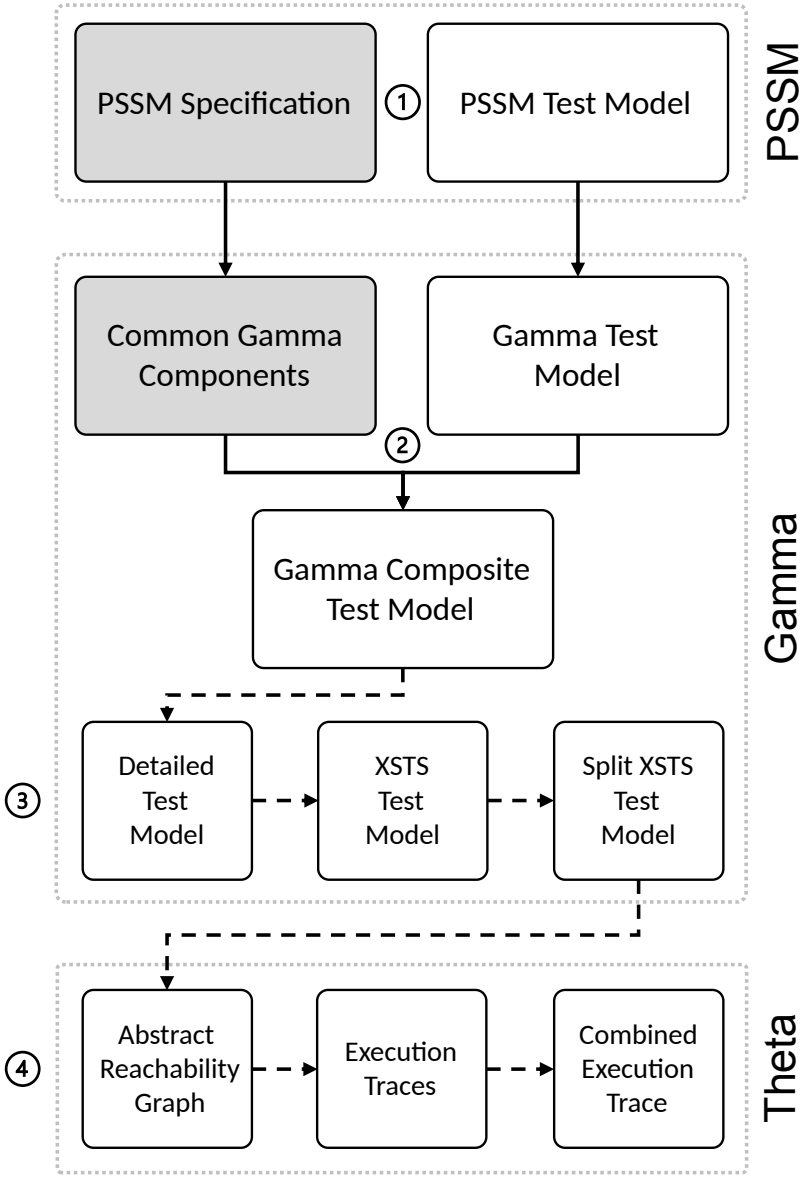


Figure 3.3: Artifacts produced during the PSSM validation workflow.

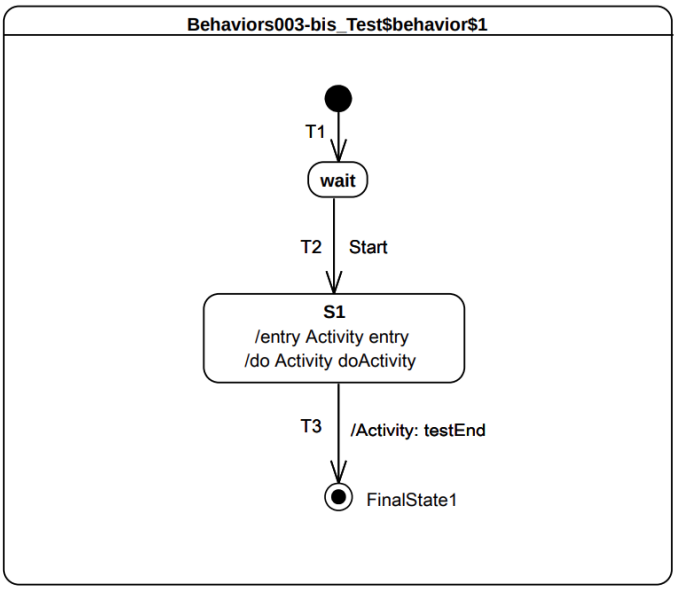


Figure 3.4: State Machine Diagram of Target component – Behavior 003-B [12] test case.

3.3 Running Example

Throughout the rest of the work, we demonstrate the different steps of the proposed workflow using an example model. In this section, we showcase the Behavior 003-B [12] test model from the PSSM Test Suite and analyze its specified valid execution trace.

3.3.1 Test Model

We have chosen Behavior 003-B [12] test case as a running example. The Target State Machine of the test is shown in Figure 3.4. After instantiation, the initial RTC step takes the State Machine into the *wait* state, in which it will stay until a *Start* signal is received. Upon receiving the Start signal, it enters *S1*, which has an entry behavior tracing `S1(entry)`. After the entry behavior has finished, the state's doActivity is started asynchronously. S1 has a completion transition to *FinalState1*, which will only fire upon successful completion of the doActivity. The test is only considered successful, if it sends the *testEnd* signal by firing transition T3.

The doActivity is shown in Figure 3.5. It begins its execution by taking a reference of *this* (which is the State Machine's context) and duplicates the value using a *fork* node. The separate branches go to two trace calls – the first one traces `S1(doActivityPartI)`, while the second one traces `S1(doActivityPartII)`. Since there is an *AcceptEventNode* between the two trace calls, the activity must receive a *Continue* signal to execute the second trace call, finish execution and thus let the statechart reach its final state.

3.3.2 Specified Trace

The PSSM specification lists a single valid execution trace for this model:

- `S1(entry)`

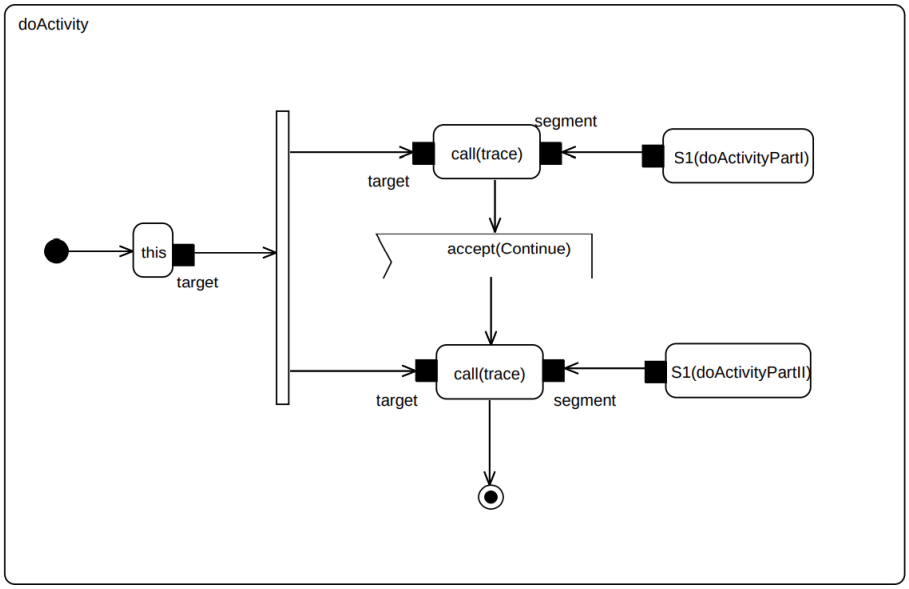


Figure 3.5: Activity Diagram of S1's DoActivity – Behavior 003-B [12] test case.

- `S1(doActivityPartI)`
- `S1(doActivityPartII)`

The specification also provides the RTC steps during execution, which is shown in Table 3.1. First, it fires transition *T1* and enters state *wait* as part of the *initial RTC step*. Afterwards, a *completion event* is generated for state *wait*, as it does not have an entry behavior. Since there is no *completion transition* from *wait*, this event is discarded. Given, that the *Start* signal is in the event pool, it triggers transition *T2*. This transition takes the State Machine into state *S1*, which asynchronously starts its doActivity Behavior. Afterwards, the State Machine receives a *Continue* signal, which is dispatched to the doActivity, enabling it to finish execution and let *S1* generate a *completion event*. Since S1 has a *completion transition*, it fires, taking the State Machine into its final state, and sending a *testEnd* signal to the Semantic Test component, thus successfully completing the test.

Step	Event pool	State Machine configuration	Fired transitions(s)
1	[]	[] - initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2]
4	[**Continue**]	[S1]	[] – doActivity RTC
5	[**CE(S1)**]	[S1]	[T3]

Table 3.1: The run-to-completion steps for an execution of the Behavior 003-B [12] test case.

3.3.3 Deadlock of Test Model

Upon closer examination of the PSSM semantics and the test case, we found that in certain situations the model can enter a deadlock[2] state before reaching its final state, which is not specified as a valid trace by the PSSM standard. The trace of that execution would look like the following:

- `S1(entry)`
- `S1(doActivityPartI)`
- **(deadlock, final state not reached)**

The RTC steps for this trace are shown in Table 3.2. The beginning steps are the same up to the point of entering state *S1*. Given the concurrent nature of doActivities, it is possible, that S1's doActivity does not reach its *AcceptEventAction*, and thus does not register any `EventAccepter` for the *Continue* signal before the *event dispatch* begins. In this case, since there are **no** event accepters for the Continue signal, it is *discarded*. After this point, the doActivity will reach its trace action and AcceptEventNode, thus registering a new event accepter for the Continue signal. Since doActivities are only considered completed when they *do not* have any event accepters registered and have no more actions to execute, the doActivity remains active, which prevents S1 from generating a completion event, thus the State Machine may never fire T3. After this point, the State Machine has no more steps to take, thus the test case will never complete (testEnd signal).

Step	Event pool	State Machine configuration	Fired transitions(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2]
4	[**Continue**]	[S1]	[]

Table 3.2: The run-to-completion steps for a deadlocked execution of the Behavior 003-B [12] test case.

Problem statement The lack of this execution trace specification in the PSSM standard allows discrepancies in interpreting UML models. For example, a PSSM conform simulator might assure a systems engineer that a similar model always terminates successfully, yet code generated by a PSSM conform code-generator might enter this deadlock state during execution. This, and other underspecifications can result in hard-to-find bugs in real systems.

[2] A state in which the State Machine does not have any more legal steps.

Chapter 4

Modeling PSSM State Machines

UML State Machines have complicated execution semantics, defined using only natural language specification. Although PSSM refined the behavioral semantics, it remains a challenge to accurately replicate said behavior, due to the sheer size of the specification and the amount of underspecified aspects [4, 3]. This chapter models a subset of the PSSM State Machine behavior using Gamma components, in the context of the PSSM Test Suite validation workflow.

The chapter is structured as follows. Section 4.1 briefly describes the changes in the Gamma AcTivity Language. In Section 4.2, we formulate the internal structure of PSSM State Machines using Gamma components. Finally, Section 4.3 showcases the interactions between the internal components during the execution of Behavior 003-B [12], providing further insight into how the model can enter a deadlock state.

4.1 Activity Component

The Gamma AcTivity Language (GATL) defined in our previous work [30] already introduced a doActivity statement, which can run activities concurrently inside states. However, in that formalism, the activity was part of the statechart. Since this work relies on activities being separate components, we formulate a new *Activity Component*.

Listing 4.1 provides an example activity component in the new syntax of the language. Activity Components are *atomic components* similar to statecharts, as they have *ports*, *internal state* and a transfer function defining its execution. The body of an activity component contains the activity nodes and flows already defined in GATL. Activity components have two special ports: the *Activity Controller Port*, which allows them to be connected to statecharts, and the *Rtc Port* which allows them to periodically reexecute themselves.

ActivityControllerPort Ports annotated with `@ActivityController` are considered *Activity Controller Ports*. Such ports enable or disable the execution of a component; by sending a *control* event with a `true` argument the activity begins execution, while a `false` argument will terminate it. The event's `isActive` parameter is *persistent*, which means the value of the parameter remains accessible throughout multiple execution cycles – in contrast to the default parameters whose values remain only for the duration of a single cycle. When the component receives an event, it is only executed if the `isActive` parameter is `true` – which means the activity has not received a *stop* event since the last *start*

```
// Interfaces provided as a library of the language
interface ActivityControllerInterface {
    in persistent event control(isActive : boolean) // argument stored between events
    out event done
}
interface Rtc {
    internal event Run // this event may only be raised by the Activity itself
}
// An example Activity component
activity Activity [
    @ActivityController port controller : provides ActivityControllerInterface
    @RtcPort port rtc : provides Rtc
    port data : provides Data // additional port for incoming events
] {
    initial init
    trigger Wait when data.continue on-await {
        log "Begun waiting";
    }
    action Done {
        log "Finished waiting";
    }
    final fin
    succession from init to Wait
    succession from Wait to Done
    succession from Done to fin
}
```

Listing 4.1: Example Gamma Activity Component showcasing the language changes.

event. The port also has a *done* event, which can be raised by the activity when it has finished execution.

RtcPort Ports annotated with `@RtcPort` are considered *RTC Ports*. Such ports provide *internal* run events, which are sent and received by the activity itself. After an execution, if the activity has any legal step left to execute, it sends itself a *run* event, causing a reexecution of the component. In doing so, a full run-to-completion of the activity is split up into multiple *sub steps* with *run* calls in between. This allows the execution to overlap with other components, modeling the behavior of *concurrent* execution.

Langauge Extensions A new `log "trace_information";` statement has been added to the Gamma Action Language, which works similarly to the `trace(string)` function used in the PSSM Test Suite. Gamma Activity *Trigger* nodes have also been extended with an `on-await { }` block, which is executed when the node is first *reached*.

Listing 4.1 shows an example activity component, which has the following behavior. Upon receiving a `control(true)` event through its activity controller port the component begins execution. Its action is the *Wait* trigger node, which *logs* the string `Begun waiting`. Since the activity has no legal next step at this point, it does not raise its run event. It will only resume execution upon receiving a *continue* event, which executes the action *Done*. The node Done has a successor node, thus the activity raises its run event. This next execution takes the activity into its *final* node. Since the activity does not have any legal next step, and there are no currently active trigger nodes, it raises its *done* event, signaling its completion to its controller statechart.

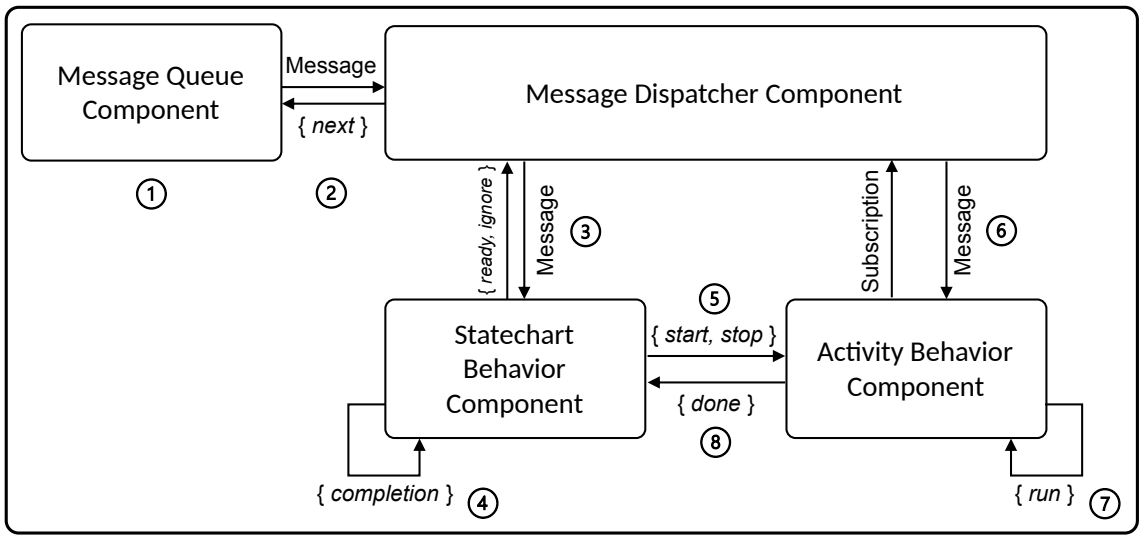


Figure 4.1: Overview of the PSSM State Machine internal structure.

4.2 Overview of the Internal Structure

As previously discussed (see Section 2.1.5) the PSSM Test Suite is composed of a Tester and a Target component, which are orchestrated by the Semantic Test component. Since we only want to trace the execution of the *Target* component, we only model the Target State Machine precisely at the time when all incoming events have arrived (but not yet processed). This results in a self-contained model, that when executed, provides the same behavior as the Target State Machine would.

Figure 4.1 depicts a Gamma asynchronous component that models the internal components of a PSSM State Machine. The different parts have been numbered for ease of understanding. ① The *Message Queue Component* represents the internal event pool of the State Machine. ② It has a connection with the main *Message Dispatcher Component*, whose purpose is to distribute incoming Messages between the internal components. They interact using a *pull* pattern, which means the Dispatcher asks for the next *Message* by sending a *next* event. ③ The Dispatcher may then forward this message to the underlying *Statechart Behavior Component*, which models the *actual* behavior of the state machine. ④ The Statechart may also send *itself* various *completion* events, to model different *state completion events*. ⑤ To accurately model the concurrent behavior of doActivities, each doActivity is represented by an *Activity Behavior Component*. The Statechart Component may *start* and *stop* them at any time by sending them a *control* event with *isActive* being *true* or *false*, respectively. ⑥ The Dispatcher may also choose to distribute incoming Messages to the Activity Components. Activities can *subscribe* to specific Message kinds asynchronously, in which case the Activity Component might receive an incoming Message of that kind from the Dispatcher – one Message for every subscription. ⑦ In order to model the interleaving of several activities, Activity Behavior Components may send themselves a *run* event, which reexecutes them in the future. ⑧ Finally, upon its completion, the Activity Component may send the Statechart a *done* event.

4.2.1 Common Interfaces

Gamma uses *interfaces* for specifying the events that can be raised on a port. Listing 4.2 shows all the interfaces used in the Gamma library of PSSM State Machines. The *Data* interface has an *event* for each *Message* kind the *Target* component may receive during a test. *QueueControl* provides the control interface for MessageQueues. *DispatcherControl* interface provides the interface for the communication between the Dispatcher and the Statechart Component. Completion events may be raised by the use of the *StatechartCompletion* interface, which has a single internal completion event. *ActivityDispatcherControl* defines the subscription communication between the Dispatcher and the Activities. It *extends* the *Data* interface with a *resetSubscriptions* event, which can be used to remove an activity's existing subscriptions.

```
1  interface Data {
2      out event start
3      out event continue
4      out event anotherSignal
5      out event pending
6      out event data(value : boolean)
7      out event integerData(value : integer)
8  }
9  interface QueueControl {
10     out event next
11 }
12 interface DispatcherControl {
13     out event ignore
14     out event ready
15 }
16 interface ActivityDispatcherControl extends Data {
17     out event resetSubscriptions
18 }
19 interface StatechartCompletion {
20     internal event completion
21 }
```

Listing 4.2: The common interfaces of the Gamma models.

4.2.2 Message Queue Component

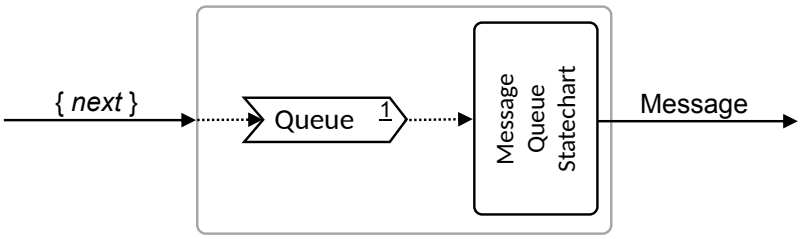


Figure 4.2: The asynchronous adapter of the Message Queue Component.

In general, Message Queue Components represent the event pools of State Machines. They *must* respond to a *next* event with the next Message to be processed. Generally, a Message Queue is modeled using a simple statechart with an asynchronous adapter component wrapping it. Figure 4.2 displays the asynchronous adapter of the Message Queue Component. It has a single incoming event, which is forwarded into the Queue.

Listing 4.3 shows the Statechart model of the Message Queue of Behavior 003-B [12]. Since the Target component is specified to receive a *start* and a *continue* event, the model of the Message Queue contains these events encoded in its structure. Upon receiving the first *next* event, it sends out the event *start*, and upon the second *next* event it sends out a *continue* event. This statechart does nothing after this point.

```
1  @Asynchronous // automatic asynchronous adapter wrapping
2  statechart MessageQueue [
3      port queueControl : requires QueueControl
4      port data : provides Data
5  ] {
6      region Main {
7          initial Initial
8          state _0
9          state _1
10         state _2
11     }
12     transition from Initial to _0
13     transition from _0 to _1 when queueControl.next / raise data.start;
14     transition from _1 to _2 when queueControl.next / raise data.continue;
15 }
```

Listing 4.3: Gamma Statechart describing the Message Queue of Behavior 003-B [12] test case.

4.2.3 Statechart Behavior Component

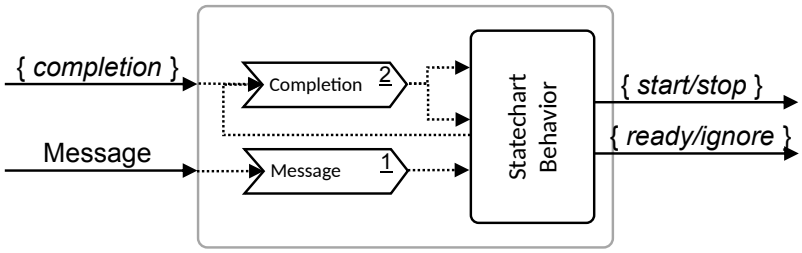


Figure 4.3: The asynchronous adapter of the Statechart Behavior Component.

In general, Statechart Behavior Components represent the actual behavior of the State Machine. They receive Messages from the Dispatcher, to which they *must* respond with either a *ready* or *ignore* event, specifying whether the statechart has consumed the incoming message or not, respectively. Statechart Behavior Components may also communicate with several Activity Behavior Components, modeling their executing doActivities. Figure 4.3 displays the asynchronous adapter of Statechart Behavior Components. A Statechart Behavior Component has two event queues, one for *completion* events, and one for the incoming dispatched messages. Completion events may come from the associated Activity Components, as well as from the statechart itself as *state completion* events. Since in UML completion events have priority over standard messages, the Completion event queue has a higher priority than the Message event queue.

As an example, the Gamma implementation of the Behavior 003-B [12] can be seen in Listing 4.4. The statecharts region schedule is changed using the `@RegionSchedule=bottom-up` annotation to follow UML semantics. The statechart has three ports, an *activity controller* for controlling S1's doActivity, an incoming *data* port for incoming Messages from the Dispatcher, and a *dispatcher control* port for responding to incoming messages.

The states of the Target State Machine are replicated in the statechart, wrapped in the *WrapperFunction* state. This is done to provide a *default ignore* response for incoming Messages. by adding an *internal* transition, that is the outmost transition, it will only be triggered, if no other transition is triggered, and does not cause exiting or entering internal states, thus does not affect the State Machine's original behavior. S1's doActivity is modeled by *raising* the associated Activity Components *control* variable with a `true` argument in its *entry* action, and with a `false` argument in its *exit* action. Transition *T2* is triggered by a Message, thus its effect is sending the Dispatcher a *ready* event, signaling that the statechart has consumed the Message. State S1 also as a *log* statement, which represents the original `trace()` function call.

```
@RegionSchedule=bottom-up // UML-style transition priority
statechart TargetSC [
    port activity_0 : requires ActivityControllerInterface
    port data : requires Data
    port dispatcherControl : provides DispatcherControl
] {
    region Wrapper {
        initial WrapperInitial
        state WrapperFunction {
            region Main {
                initial Initial
                state wait
                state S1 {
                    entry / log "S1_entry"; raise activity_0.control(true);
                    exit / raise activity_0.control(false);
                }
                state FS1
            }
        }
    }
    transition from WrapperInitial to WrapperFunction
    // Transition sendding an ignore event if no other transition triggered
    @Internal transition from WrapperFunction to WrapperFunction when data.any /
        raise dispatcherControl.ignore;

    transition "T1" from Initial to wait
    // Since T2 is triggered by a Message, it raises the ready event
    transition "T2" from wait to S1 when data.start / raise dispatcherControl.ready;
    transition "T3" from S1 to FS1 when activity_0.done
}
```

Listing 4.4: Gamma Statechart describing the internal behavior of Behavior 003-B [12] test case.

Listing 4.5 shows the asynchronous adapter of the statechart, specifying the two event queues. Since the statechart may generate and receive multiple completion events from the various states and doActivities, the *completionQueue*'s capacity is greater than 1. `QUEUE_SIZE` is a constant value defining the common queue sizes of the models. Since the Dispatcher will wait for a *ready* or *ignore* Message from the Statechart Component, the *dataQueue* can stay at the capacity of 1.

4.2.4 Activity Behavior Component

The Activity Behavior Component represents the instances of doActivities executed in the State Machine. Figure 4.4 shows the asynchronous adapter of the Activity Behavior Component. Activities have multiple event queues, whose priority determines which incoming events are processed first. Control events have the highest priority, in order to guarantee the Activity always responds to *start* and *stop* events, thus an activity will never run after

```
adapter Target of component target : TargetSC {
    when any / run
    queue completionQueue(priority = 2, capacity = QUEUE_SIZE) {
        activity_0.any
    }
    queue dataQueue(priority = 1, capacity = 1) {
        data.any
    }
}
```

Listing 4.5: Asynchronous adapter component of the Statechart Behavior Component of Behavior 003-B [12] test case.

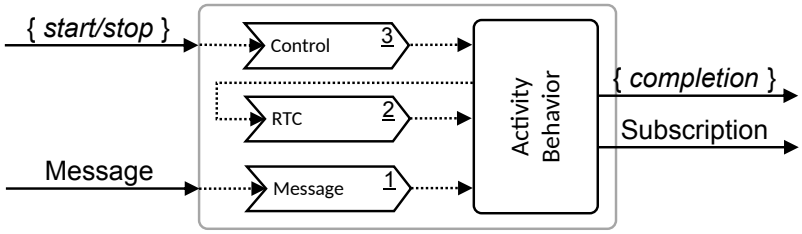


Figure 4.4: The asynchronous adapter of the Activity Behavior Component.

a stop event. Since PSSM prohibits incoming Message processing during an RTC step, the RTC event queue has the second highest priority.

As an example, the Gamma implementation of the doActivity of S1 in Behavior 003-B [12] can be seen in Listing 4.6. Additionally to the activity ports already described, it also has a *dispatcher control* port and a *data* port, which allow the activity to subscribe to and receive Messages. This activity begins execution by *logging* the word `S1\_doActivityPartI`. Afterwards, it executes the *Wait* trigger node, which *subscribes* to the *continue* event. After receiving a *continue* event from the Dispatcher, its execution completes by logging the word `S1\_doActivityPartII`. Finally, it sends a *done* event to the executing Statechart Component.

Listing 4.7 shows the asynchronous adapter of the Activity Component, specifying the three event queues. Since Statechart Components may send multiple *control* events to the Activity Component (e.g, first starting it then stopping it) the new incoming events must override the previous ones: `discard = oldest` sets this behavior for the *controlMessages* event queue. Due to their concurrent behavior, Activity Components may receive multiple Messages between executions, thus their the *dataMessages* event queue has also a bigger capacity. Since the *run* event in the *rtcMessages* queue is only used to reexecute the component, its capacity can remain 1.

4.2.5 Message Dispatcher Component

The Message Dispatcher Component represents the *dispatch* functionality of UML objects and is the main driver of the interactions inside the State Machine. An excerpt of the Gamma implementation is displayed in Listing 4.8. At initialization, it sends a *next* event to the *Message Queue*, signaling that it requires the next *Message*. Upon receiving the Message, it has a choice to make: it can either send the Message to the Statechart Component or to one of the Activity Components.

```
activity Activity [
    @ActivityController port controller : provides ActivityControllerInterface
    @RtcPort port rtc : provides Rtc
    port dispatcherControl : provides ActivityDispatcherControl
    port data : requires Data
] {
    initial init
    action Part1 {
        log "S1_doActivityPartI";
    }
    trigger Wait when data.continue on-await {
        raise dispatcherControl.continue;
    }
    action Part2 {
        log "S1_doActivityPartII";
    }
    final fin

    succession from init to Part1
    succession from Part1 to Wait
    succession from Wait to Part2
    succession from Part2 to fin
}
```

Listing 4.6: Gamma Activity describing the doActivity run in S1 of Behavior 003-B [12] test case.

```
adapter DoActivity of component act : Activity {
    when any / run

    queue controlMessages(priority = 3, capacity = 1, discard = oldest) {
        controller.any
    }
    queue rtcMessages(priority = 2, capacity = 1) {
        rtc.any
    }
    queue dataMessages(priority = 1, capacity = QUEUE_SIZE) {
        data.any
    }
}
```

Listing 4.7: Asynchronous adapter of the Activity describing the doActivity run in S1 of Behavior 003-B [12] test case.

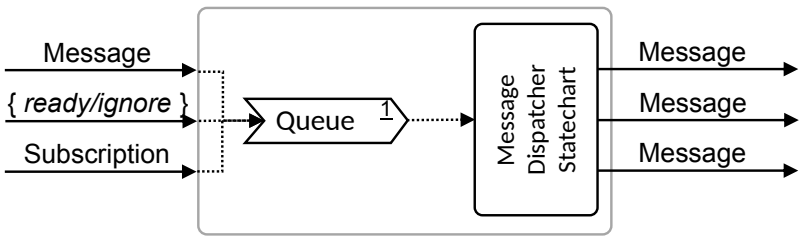


Figure 4.5: The asynchronous adapter of the Message Dispatcher Component.

```
statechart DispatcherSC [
    // MessageQueue ports
    port queueControl : provides QueueControl
    port inputData : requires Data
    // Statechart Behavior ports
    port dispatcherControl : requires DispatcherControl
    port outputData : provides Data
    // Activity Behavior ports
    port dispatcherControlActivity_0 : requires ActivityDispatcherControl
    port outputDataActivity_0 : provides Data
] {
    // variables tracking the number of subscriptions for a given Activity Component
    var activity_0_Start : integer := 0
    // ...
    var activity_0_IntegerData : integer := 0
    // variables storing the value of send Message parameters
    var sentIntegerDataValue : integer := 0
    region Main {
        initial Initial
        state WaitingForEvent {
            entry / raise queueControl.next; // sending 'next' to messageQueue
        }
        state WaitingForTarget {
            region Sent {
                // ...
                state Sent_Start
                choice DecideActivity_Start
                merge MergeActivity_Start
                // ...
                state Sent_IntegerData
                choice DecideActivity_IntegerData
                merge MergeActivity_IntegerData
            }
        }
    }
    region EventSubscription {
        initial EventSubscriptionInitial
        state ListeningToSubscriptions
    }
    transition from Initial to WaitingForEvent
    // dispatching incoming start event
    transition from WaitingForEvent to WaitingForEvent when inputData.start [activity_0_Start>0]
      /  raise outputDataActivity_0.start; activity_0_Start:=activity_0_Start-1;
    transition from WaitingForEvent to Sent_Start when inputData.start / raise outputData.start;
    transition from Sent_Start to DecideActivity_Start when dispatcherControl.ignore
    transition from DecideActivity_Start to MergeActivity_Start [activity_0_Start>0] /
        raise outputDataActivity_0.start; activity_0_Start:=activity_0_Start-1;
    transition from DecideActivity_Start to MergeActivity_Start [else];
    transition from MergeActivity_Start to WaitingForEvent
    // dispatching incoming events ...
    // statechart processed this event
    transition from WaitingForTarget to WaitingForEvent when dispatcherControl.ready
    // processing event subscriptions
    transition from ListeningToSubscriptions to ListeningToSubscriptions when
     dispatcherControlActivity_0.start / activity_0_Start:=activity_0_Start+1;
    // ...
    transition from ListeningToSubscriptions to ListeningToSubscriptions when
     dispatcherControlActivity_0.integerData / activity_0_IntegerData:=activity_0_IntegerData+1;
}
```

Listing 4.8: Gamma Statechart of all Dispatcher Componenents.

- If there is an Activity Component that has subscribed to the incoming Message previously, the Dispatcher *may*[1] choose to dispatch the Message to that component. Given, that an Activity Component will only subscribe to a Message if it can process it (Section 4.2.4), the Dispatcher can consider this as a successful dispatch, and move on to the next Message.
- The Dispatcher may choose to try to send the Message to the Statechart first, in which case it forwards the Message, and stores its specific details in the current state. If the statechart responds with a *ready* event, then it has processed it, thus the dispatcher may move on to the next Message. Otherwise, it will try to resend it to one of the Activity Components using the stored details. If none of the above succeeds then the Message is *discarded*.

This non-deterministic choice is modeled using *conflicting* transitions inside the behavior model. Initially, the model starts in the *WaitingForEvent* state, which automatically sends out a *next* event to the Message Queue. There are multiple conflicting transitions from this state for each possible incoming Message: one for the statechart, and one for each connected activity *guarded* with if it has subscribed to the specific Message (`[activity_0_Start>0]`). Since the Dispatcher has to wait for the reply event (ignore or ready) from the Statechart Component, it must store the sent Message in its state. In the case of *Start* Message, the *Sent_Start* state is entered, from which there are multiple ways, to return: if there is an Activity Component subscribed to the Start Message then it is dispatched to one of them, otherwise, it discards the Message. In either case, it transitions to the *WaitingForEvent* state, starting the whole process of asking for the *next* Message and trying to dispatch it to one of the components over again.

In order to track Message subscriptions, the Dispatcher has an orthogonal region with a single state with transitions triggered by each subscription Message. The number of subscriptions is tracked with an integer value for each Activity Component. This variable is decremented when the dispatcher sends Messages to the Activity Component since it is no longer waiting for that Message, and incremented if the Activity Component subscribes to the specific Message.

Listing 4.9 shows the asynchronous adapter of the Dispatcher Component, specifying the three event queues. Since the Dispatcher may receive multiple subscriptions from multiple executing Activity Components, its *activityControlMessages* capacity is increased from 1. As the Message Queue and the Statechart Behavior components only send one-one events at a time, their event queues can remain at the capacity of 1. In order to allow Activity Components to send subscription events concurrently to the Statecharts behavior, the *activityControlMessages* event queue's priority has to be the highest. The priority of the other two event queues does not matter, as no events can occur concurrently from the Message Queue and the Statechart Behavior components.

4.2.6 Constructing the State Machine Component

Finally, since the inherent behavior of PSSM State Machines is asynchronous in nature, the whole Gamma Test Model is wrapped using an *async* component. Listing 4.10 shows the Gamma implementation of the Behavior 003-B [12] system, instantiating the different components, and connecting them using channels. Since the model does not have any environment, the *System* component does not expose any ports.

[1] This is a non-deterministic choice.

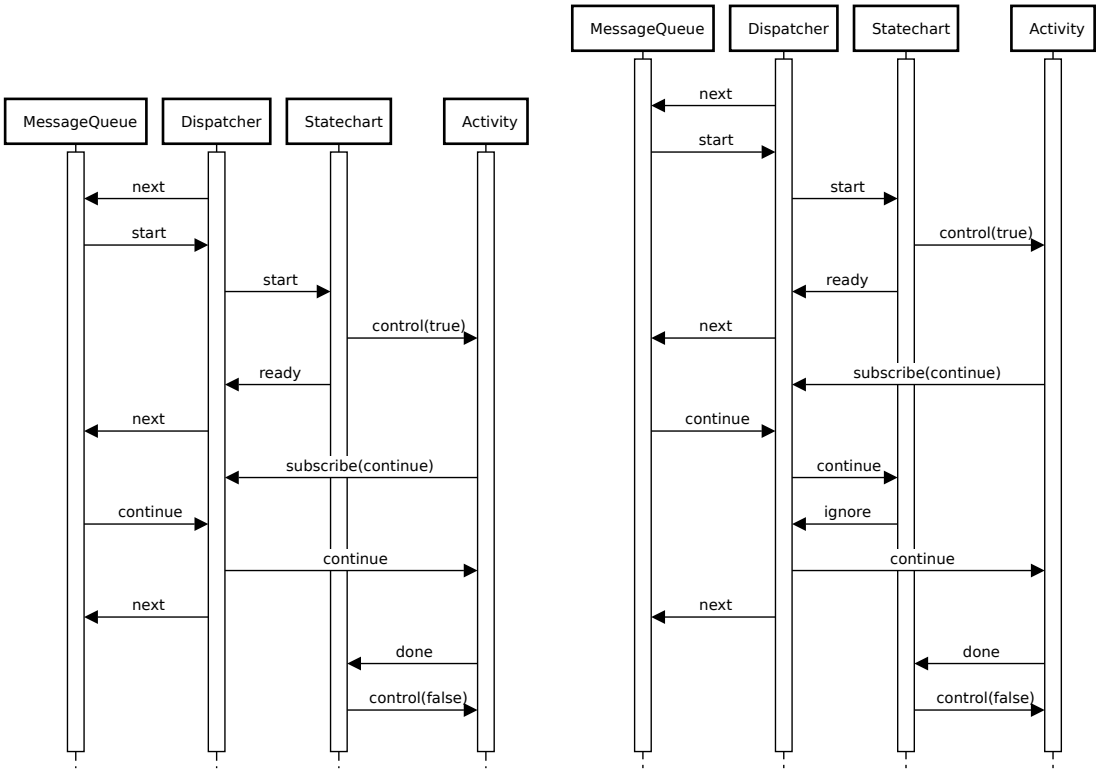
```
adapter Dispatcher of component dispatcher : DispatcherSC {
    when any / run

    queue activityControlMessages(priority = 3, capacity = QUEUE_SIZE) {
        dispatcherControlActivity_0.any
    }
    queue controlMessages(priority = 2, capacity = 1) {
        dispatcherControl.any
    }
    queue dataMessages(priority = 1, capacity = 1) {
        inputData.any
    }
}
```

Listing 4.9: Asynchronous adapter of the Dispatcher Component.

```
async System {
    component messageQueue : MessageQueue
    component target : Target
    component dispatcher : Dispatcher
    component doActivity_0 : DoActivity
    // Connecting messageQueue to dispatcher
    channel [ dispatcher.queueControl ] -o)- [ messageQueue.queueControl ]
    channel [ messageQueue.data ] -o)- [ dispatcher.inputData ]
    // Connecting dispatcher to target
    channel [ dispatcher.outputData ] -o)- [ target.data ]
    channel [ target.dispatcherControl ] -o)- [ dispatcher.dispatcherControl ]
    // Connecting doActivity_0 to target
    channel [ doActivity_0.controller ] -o)- [ target.activity_0 ]
    // Connecting doActivity_0 to dispatcher
    channel [ doActivity_0.dispatcherControl ] -o)- [ dispatcher.dispatcherControlActivity_0 ]
    channel [ dispatcher.outputDataActivity_0 ] -o)- [ doActivity_0.data ]
}
```

Listing 4.10: The Gamma Test Model of Behavior 003-B [12] test case.



(a) *Continue* dispatched to the Activity. **(b)** *Continue* dispatched to the Statechart first.

Figure 4.6: Interactions during successful execution of Behavior 003-B [12].

4.3 Interaction Example

By modeling the State Machine with multiple internal components that interact with each other asynchronously, we may gain a deeper insight into the underlying mechanism of PSSM State Machines. Figure 4.6 shows two distinct sequences of interactions between the internal components of the constructed Target State Machine of Behavior 003-B [12] (introduced in Section 3.3), that results in the system reaching its final state. In Figure 4.6a the interaction starts with the Dispatcher asking for the next Message. Upon receiving it, since no Activity has subscribed to the *start* message, it dispatches it to the Statechart. The Statechart has a trigger for the *start* message, which takes it into state S1, *starting* its doActivity, and replying with a *ready* to the Dispatcher. The Dispatcher asks for the next Message, which is a *continue* Message. Meanwhile, the activity is running *concurrently* to the components, and has *subscribed* to the continue Message. Since there is a subscriber to the continue Message, the Dispatcher *chooses* to dispatch it to the Activity, and proceeds to ask for the next Message. Since the MessageQueue has run out of Messages, it will never receive a reply. Meanwhile, the Activity has finished execution and sent a *done* event to the Statechart, which in result executes its completion event, stopping the Activity and finishing the test successfully. The interaction shown in Figure 4.6b differs in that the Dispatcher first tries to dispatch the continue Message to the Statechart, which responds with an *ignore* event, thus the Dispatcher retries with the Activity. Note, that in order for the Dispatcher to *be able* to dispatch the continue Message to the Activity, it had to receive the subscription *before* the dispatching process finished.

Figure 4.7 shows an interaction, in which the State Machine reaches the deadlock state described in Section 3.3.3. The execution starts similarly to the ones described above, only differing in the *timing* of the subscribe event being sent to the Dispatcher. Since it arrived *after* the dispatching process finished, the incoming Message has already been *discarded*, hence the Activity will never be able to resume execution.

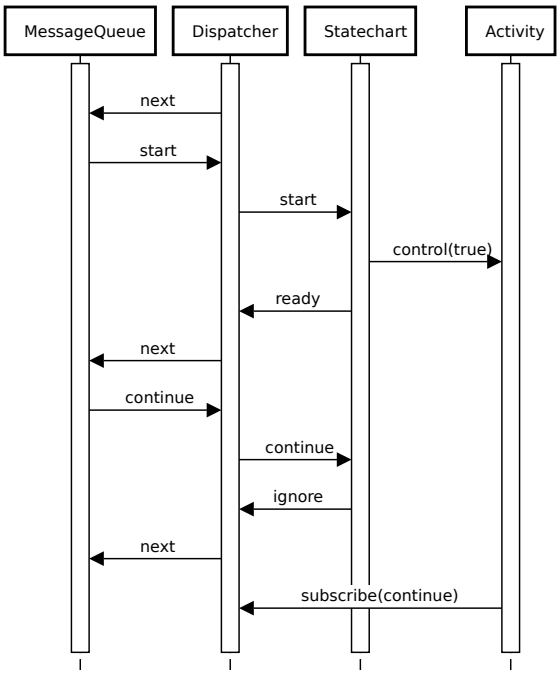


Figure 4.7: Interactions during execution of Behavior 003-B [12] reaching a deadlocked state.

Chapter 5

Systematic Generation of Precise Execution Traces

In order to provide the reviewers and experts with validation insights during the validation workflow, it must provide an exhaustive and precise set of execution traces for the test models. As Gamma models have formal semantics, we can use formal methods to generate a complete set of execution traces.

The chapter is structured as follows. Section 5.1 describes how models trace their execution. Section 5.2 introduces the XSTS split algorithm. Section 5.3 describes the algorithm that explores and records the available execution traces exhaustively. Finally, Section 5.4 showcases the trace generated for the Gamma implementation of Behavior 003-B [12].

5.1 Tracing Information

Gamma uses automatic semantic-preserving model transformations to transform Gamma models into the XSTS formalism – which is one of the input languages of Theta. By extending this tool chain, we are able to construct execution traces that *precisely* follow the execution semantics of the original Gamma models. We extended Theta with a *Simulator* component, that is able to exhaustively traverse all possible executions of an XSTS model and record changes in the variables. By specifying which variables to *track*, we are able to reconstruct the execution traces.

Gamma log statements serve the purpose of tracing information during a model's execution. During the Gamma $\rightarrow$ XSTS transformation, for every component type C of the Gamma model, an enum type D_{log_C} is defined in the XSTS model, with literals corresponding to the string inputs of log statements in C. For every component instance c of type C, a log variable $log_c \in V$ is defined in the XSTS model $\langle V, Tr, In, En \rangle$, with domain $D_{log_c} = D_{log_C}$. Then, the Gamma log statements of form $log(\varphi)$ in component c are modeled as $log_c := \varphi$ assignments in XSTS. During a Gamma model's execution, we are able reconstruct its execution trace by tracking variables log_c.

5.2 Splitting XSTS Models

Gamma models transformed into XSTS represent one step of their execution as a single monolithic XSTS transition. This transition may have several sequence and non-

deterministic operations. To execute the Gamma model, this XSTS transition must be executed over and over again. The granularity of XSTS transitions has a serious effect on the verification of such systems in terms of required time and memory. This effect is investigated in our previous work [26], and measurements showed that splitting monolithic transitions into smaller ones causes a huge overhead on model checking – this design decision behind the model transformation in Gamma remains justified.

However, since XSTS transitions are atomic in nature, the simulator could lose tracing information during execution (e.g., two log statements on a single transition effect). Also, any non-determinism inside this monolithic transition would be lost by the simulator. In order to detect every log statement, and be able to control the non-deterministic choices made during execution, the monolithic transition must be split into multiple deterministic parts, as presented in our previous work [26].

We extended the splitting algorithm of XSTS transitions with handling *parallel* operations, in order to explore every possible execution overlap of several parallel branches. This work does not require the detailed definition of the splitting algorithm, however, since precise semantics is essential when using formal methods, Appendix A provides a detailed formulation.

Briefly, splitting is a semantics-preserving model transformation, which transforms an XSTS model into the form $\langle V, Tr, In, En\rangle \rightsquigarrow \langle V', Tr', In', En'\rangle$ where $In' = En' = \emptyset$, $V \subseteq V'$, $|Tr'| \geq |Tr| + |In| + |En|$, and every $t' \in Tr'$ transition is deterministic, and contains only a single log variable change.

5.3 Generating Execution Traces

In order to generate every possible execution trace systematically, we need to traverse the entire state space of the given XSTS model precisely. To prevent multiple states from overlapping with each other, we track the value of every variable explicitly, instead of using any kind of abstraction. Informally, from a specific state of our system, we calculate the possible successor states, until there is no more fireable transition. When we reach the end of an execution, we save its trace, then we backtrack to the last decision point where unexplored decisions remained, and continue in a new direction. This process is continued until there are no more unexplored parts of the system. We introduce *Reachability Graph* to represent the state space during the algorithm.

Theta transforms every input formalism into a common internal representation, *Abstract Reachability Graph* (ARG) [14]. ARG is used to represent an abstract state space with abstract domains, but in this work, we avoid the usage of abstraction, thus, RG is a simplification of ARG.

Definition 2 (Reachability Graph). A *reachability graph* (RG) is a 3-tuple $RG = \langle N, E, C\rangle$ where:

- N is the set of *nodes*, each $n \in N$ representing a concrete state c of the system, marked $c(n) = c$.
- $E \subseteq N \times Ops \times N$ is the set of *edges* labeled with *operations*. An edge $(n_1, op, n_2) \in E$ is present if $c(n_2)$ is a successor state of $c(n_1)$ with operation op.
- $C \subseteq N \times N$ is the set of *covered-by edges*. A covered-by edge $(n_1, n_2) \in C$ is present, if $c(n_1) \sqsubset c(n_2)$. Note, that without abstraction, $c_1 \sqsubset c_2 \equiv c_1 = c_2$. ▪

A node $n \in N$ is *expanded*, if all of its successors are included in RG. A node n is *covered*, if a covered-by edge $(n, n') \in C$ exists for an other node $n' \in N$.

5.3.1 Traversing Every Possible Execution

In order to explore every possible execution, we traverse the entire state space of the model. To do so, starting from the root node representing initial concrete state c_0, we build a *reachability graph* RG, until it can be expanded. As a result, we will have a complete RG, in which every *path* starting from the root node corresponds to an *Execution* of the system.

Definition 3 (Path). We define a *path* σ_P in the reachability graph $RG = \langle N, E, C \rangle$ as $\sigma_P = [n_0, e_1, n_1, ..., e_n, n_n]$ an alternating sequence of *nodes* and *edges* of the RG, where every $n_i \in N$ and every $e_i \in E$. ▪

Definition 4 (Execution). We define an *execution* σ of a split XSTS model $\langle V, Tr, In, En \rangle$, where $In = En = \emptyset$, as an alternating sequence of *concrete states* and *operations* $\sigma = [c_0, op_1, c_1, \ldots, op_n, c_n]$, where every $c_i \in C = \times_{v \in V} D_v$ is a concrete state of the model, and every op_i is the operation of a split transition (fragment) $t \in Tr$. ▪

Generally, in the case of a sequence σ, we use the notation $\sigma \leftarrow [\sigma, a]$ for adding a to the end of σ.

Note, that a path $\sigma_P = [n_0, e_1, n_1, ..., e_n, n_n]$ represents exactly one execution $\sigma = [c_0, op_1, c_1, \ldots, op_n, c_n]$, where every c_i is the concrete state represented by RG-node n_i, and every op_i is the operation of RG-edge e_i. In the following, we present an algorithm to collect every path σ_P of the RG, then map the paths to executions.

The execution traversal algorithm is presented in Algorithm 1. Every path in $RG = \langle N, E, C \rangle$ represents an execution of the system. We expand the RG until any of its nodes $n \in N$ can be expanded – a node $n \in N$ can be expanded if it has not been expanded earlier, and it is not covered by any other node $n' \neq n \in N$, i.e. no covered-by edge $(n, n') \in C$ is present. In other words, we stop expanding a path $\sigma_P = [n_0, e_1, n_1, \ldots, e_n, n_n]$, if the node $n_n \in N$ corresponding to the last state c_n of the execution is covered by another node (so the rest of the path is already discovered in a previous one), or if it has no successor nodes (so the path can not be continued).

After a path $\sigma_P = [n_0, e_1, n_1, \ldots, e_n, n_n]$ can not be further expanded, we save it to the set of paths Σ_P, and backtrack to the last node $n_{i_{max}}$, where $n_{i_{max}}$ has at least one unexpanded successor node $n'_{i_{max}+1}$, available by edge $e'_{i_{max}+1} = (n_{i_{max}}, op, n'_{i_{max}+1})$ from $n_{i_{max}}$. Then, we continue with the expansion of $n'_{i_{max}}$, resulting in a new path $\sigma'_P = [n_0, e_1, n_1, \ldots, e_{i_{max}}, n_{i_{max}}, e'_{i_{max}+1}, n'_{i_{max}+1}, \ldots]$.

After this, every node $n \in N$ is expanded, and every path $\sigma_P \in \Sigma_P$ has a final node n_n with no successor nodes. Note, that n_n may be covered by another node n' if a covered-by edge $(n_n, n') \in C$ is present. In this case, path σ_P is not a complete path, but it must be continued by any other subpath, starting from node n'.

σ_{P_last} denotes the last element (node) of σ_P. $\Sigma_P \leftarrow \Sigma_P \cup \{copy(\sigma_P)\}$ denotes that the later modification of σ_P does not change its previously created copy in Σ_P. $UnexpandedSuccessors(n, RG)$ where $n \in N$ is an RG-node, returns the set of unexpanded successors $n' \in N$ of n, i.e. the unexpanded nodes n' for which an edge $(n, op, n') \in E$ is present.

Algorithm 1: TRAVERSEEXECUTIONS Traversing every execution of a split XSTS model.

Input: Split XSTS model $XSTS = \langle V, Tr, In, En \rangle$ with initial state c_0
Output: The set of executions $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$

```
1  TraverseExecutions (XSTS)
2      Σ_P ← ∅, σ_P ← [ ]
3      N ← {n(c_0)}, E ← ∅, C ← ∅
4      RG ← ⟨N, E, C⟩
5      successors ← N
6      traverse ← ⊤
7      while traverse do
8          while |successors| > 0 do
9              node ← successor ∈ successors
10             σ ← [σ, (σ_{P_last}, op, node) ∈ E, node]
11             CLOSE(node, RG)
12             successors ← ∅
13             if node is not covered then
14                 successors ← EXPAND(node, RG)
15             end
16         end
17         Σ ← Σ ∪ {copy(σ)}
18         if ∃n_{i_max} ∈ σ_P with unexpanded successors then
19             successors ← UnexpandedSuccessors(n_{i_max}, RG)
20         else
21             traverse ← ⊥
22         end
23     end
24     Σ ← Σ_P as executions
25     return Σ
```

The RG is built by the CLOSE and EXPAND methods. CLOSE(n, RG) checks whether the given node $n \in N$ of RG can be covered with another $n' \in N$ node. If yes, it adds the corresponding covered-by edges (n, n') to C. EXPAND(n, RG) expands the RG with every successor node n', each representing a state c' which is a successor state of $c(n)$, i.e. a transition $t = (c(n), c')$ exists. For every n' node, an edge (n, op_t, n') is also added to E where op_t is the operation of transition t.

5.3.2 Representing an Execution as an Execution Trace

Instead of saving every concrete state of a concrete execution as a trace, we just track some of the variables of the system $V_T \subseteq V$. For a specific execution of the model, we would like to observe the value changes of the tracked variables in order.

Definition 5 (Execution Trace). We define an *execution trace* ET as a sequence of sets of pairs (v_T, φ), where $v_T \in V_T$ is a tracked variable and $\varphi \in D_{v_T}$ is the value of v_T from its domain D_{v_T}. An element ET_i of the trace represents the set of variables (and their values) that have changed as a result of the execution of an operation. ▪

Note, that we can observe the precise order of every value change in the tracked variables by splitting every assignment to a tracked variable into a separate fragment. To achieve

this, we just need to modify the splitting rule of sequences, by defining these assignments as splittable operations.

If we would like to observe consecutive $v_T := \varphi$ assignments, i.e. an $ET = [\ldots, \{(v_T, \varphi)\}, \{(v_T, \varphi)\}, \ldots]$, we need to introduce $v_T := \epsilon$ assignments between them where $\epsilon \in D_{v_T}$ is an unused value of domain D_{v_T}. It will result in an $ET = [\ldots, \{(v_T, \varphi)\}, \{(v_T, \epsilon)\}, \{(v_T, \varphi)\}, \ldots]$, from which, then we need to remove every $\{(v_T, \epsilon)\}$, resulting in $ET = [\ldots, \{(v_T, \varphi)\}, \{(v_T, \varphi)\}, \ldots]$.

At the initial concrete state c_0 (where every variable $v \in V$ has its initial value $c_0(v) = IV(v)$, we add the pair of every tracked variable $v_T \in V_T$ and its initial value $IV(v_T)$ into the execution trace $ET \leftarrow [\bigcup_{v_T \ in V_T}(v_T, IV(v_T))]$.

During the execution, in every concrete state c, we check whether the value of any tracked variable $v_T \in V_T$ has changed. If so, we add these *value changes* into ET. Formally, $ET \leftarrow [ET, \bigcup_{\forall v_T \in V_T:\ last(v_T) \neq c(v_T)}(v_T, c(v_T))]$ where $last(v_T)$ denotes the last value of v_T saved to ET. At the end of the execution, this algorithm will produce a list of every value change of every tracked variable.

By choosing the right set of tracked variables V_T, we can precisely control the granularity of the execution traces, i.e. the observable state changes of the system.

5.3.3 Merging Execution Traces Graphically

We found, that the most convenient way to summarize every different execution of a model is by choosing the correct set of tracked variables V_T and visualizing the execution traces as a graph.

This representation is brief but complete: it shows the *differences* of the executions in an intuitive way. Transforming an execution trace into a graph is quite intuitive, so we leave its formal definition out. Informally, the *value change sets* are transformed into nodes, and the consecutive ones are connected with directed edges.

The last node of every execution trace of executions, finishing with a covered node, is connected to the successor value changes of the covering node. As a result, the possible ends of incomplete execution traces are also shown, i.e. on this graph, every path ends in a final state of the model, regardless of the covered-by edges.

For a more compact representation, semantically the same nodes are merged into each other. Starting from the leaves, we merge two nodes n_1, n_2, if they represent the same value changes, and the sets of their successor nodes S_1, S_2 are semantically the same, recursively. Two sets of nodes S_1, S_2 are semantically the same, if $|S_1| = |S_2|$, and a mutually exclusive mapping exists between their semantically same elements.

5.4 Example Generated Trace

As an example, in this section we showcase the generated execution traces for the Behavior 003-B [12] test case introduced in Section 3.3. We used the model constructed in Section 4.2 and tracked two sets of variables to demonstrate the configurability of the trace generator, and demonstrate how well it showcases the possible interactions between the internal components.

Figure 5.1 shows the generated trace by tracking only the Target logs: $V_{T_1} = \{v_{log_{target}}, v_{log_{doActivity}}\}$. The generator also found the execution trace of reaching the *dead-*

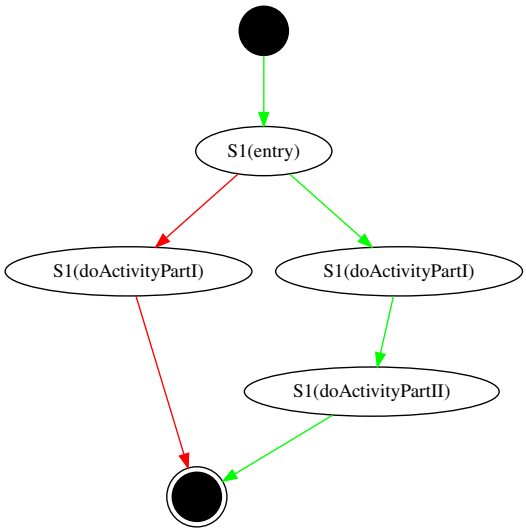


Figure 5.1: The actual execution traces of Behavior 003-B [12], found by our approach with tracking variables V_{T_1}. The expected execution trace defined by PSSM is colored green, while the execution trace reaching a deadlocked state (discussed in Section 3.3.3) is colored red.

lock state. This trace is refined in Figure 5.2, which shows the generated trace by tracking the Target logs and the Dispatcher logs as well: $V_{T_2} = V_{T_1} \cup \{v_{log_{dispatcher}}\}$.

Discussion The fact, that the trace generator found the deadlock execution using only the log statements already present in the model (i.e., no automatic detailing of the model) demonstrates the value of this approach, However, as the refined execution shows, there are various other execution traces as well, which were *abstracted* away and not shown in the simple trace. However, too much detail could have also easily overwhelmed the execution trace, essentially making it next to impossible to meaningfully analyze. Thus, it is important to view the system through different execution traces with various granularities, tracking different components, and injecting additional detailing log statements.

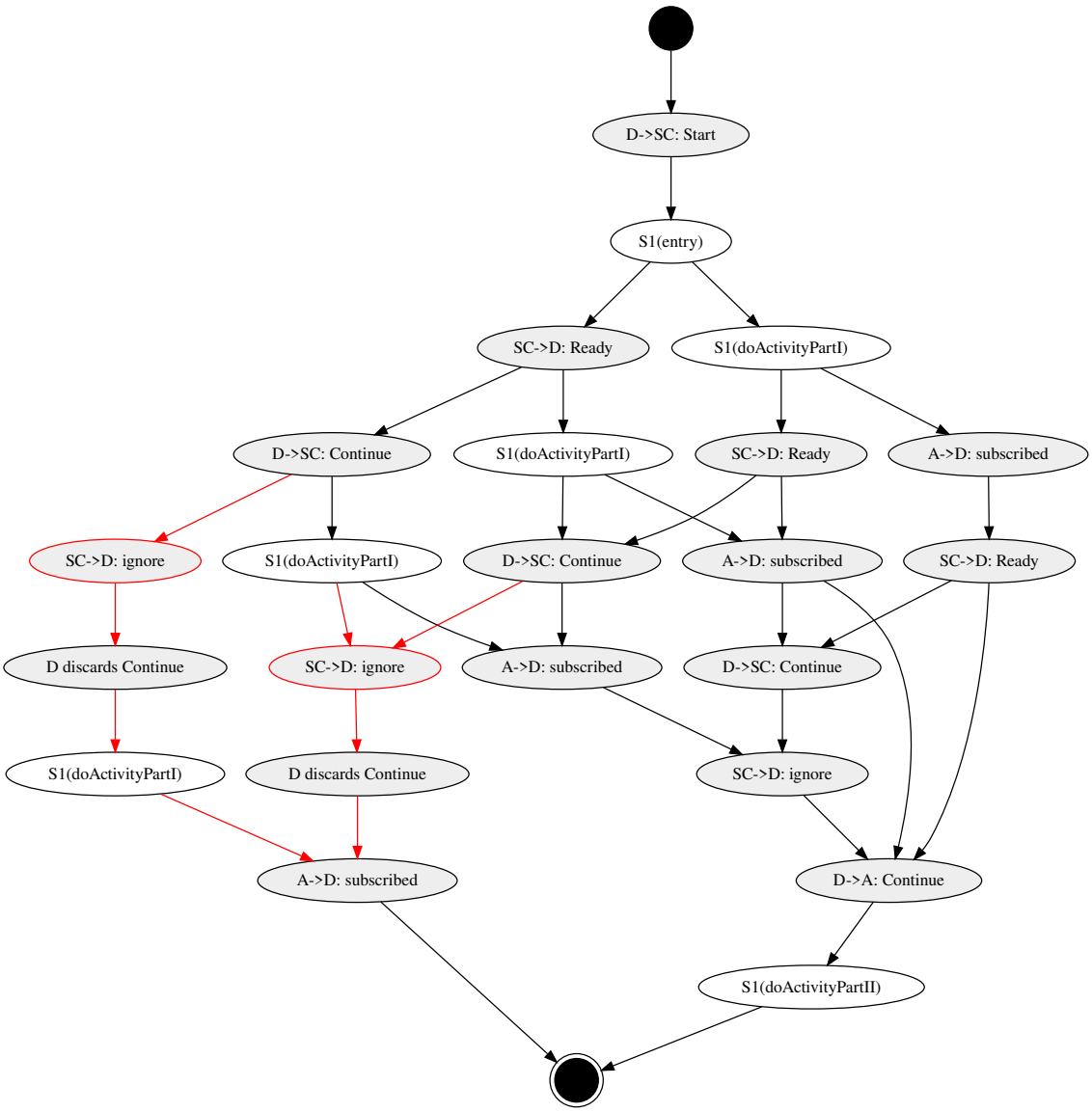


Figure 5.2: The actual execution traces of Behavior 003-B [12], found by our approach with tracking variables V_{T_2}. The nodes representing dispatcher interactions (discussed in Section 4.2) are filled with gray, in their label D denotes the dispatcher, SC denotes the statechart, and A denotes the activity. The deadlocked execution traces (discussed in Section 3.3.3) are colored with red. The red nodes show the reasons for the deadlocks, i.e. that the statechart sends the ignore message to the dispatcher before the subscription of the activity.

Chapter 6

Evaluating the PSSM Validation Workflow

Using the mapping rules from PSSM State Machines to Gamma (introduced in Chapter 4) and the execution trace generation (introduced in Chapter 5), we validated a subset of the PSSM Test Suite by modeling the Test Cases and comparing the generated traces to the ones specified by the PSSM standard. In this chapter, we systematically evaluate the constructed PSSM validation workflow and summarize our findings.

This chapter is structured as follows. Section 6.1 provides an overview on our technical contributions to the Gamma and Theta tools. In Section 6.2, we define goals for this evaluation. Section 6.3 details the constructed test model library we used as formal models. In Section 6.4, we provide an overview on the validation results. Finally, Section 6.5 summarizes the evaluation of the validation workflow.

6.1 Implementation

The implementation of the PSSM validation workflow entailed the extension of the Gamma Statechart Composition Framework and the Theta Model Checking Framework tools. By choosing a mature and well-tested tool chain as the bases of our work, our software components also inherit their capabilities, and our work becomes available for everyone else using these tools. Our implementation is open source, and will be contributed back to the tools' main GitHub repositories.

This section gives a brief overview of the implementation details of our theoretical contributions introduced in Chapter 4 – 5.

6.1.1 Contributions to the Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework[1] is a collection of *Eclipse* plugins building upon several technologies: *Eclipse Modeling Framework* (EMF) to create the metamodels, *Xtext* to define the language grammars, and *Viatra* for model transformations.

[1] `https://github.com/ftsrg/gamma`

Language Extensions

As part of this work, we extended several Gamma languages.

- We added *log statements* to the *Gamma Action Language* in order to add tracing information for models.
- We extended the *Gamma Statechart Language* with the *Internal* transition formalism of UML State Machines.
- We extended our previously proposed *Gamma AcTivity Language* [30] to allow its usage as Components.
- We extended the *Gamma Composite Language* with the new Gamma Activity Component formalism.

Gamma Transformation Extensions

As part of this work, we enhanced the Gamma transformation components to support the PSSM validation workflow.

- To support the execution trace generation, we added a new *Log Statement Injector* module to the framework, which can inject log statements into Gamma models during model transformation, extending the resulting model with additional detailed tracing information.
- We extended the Gamma $\rightarrow$ XSTS transformer component with the new language formalisms mentioned above, generating semantically correct XSTS models.
- We extended our previous work [26] of splitting XSTS operations with the splitting of *parallel actions*, thus providing fully concurrent behavior.

6.1.2 Contributions to the Theta Model Checking Framework

The Theta Model Checking Framework[2] is a *modular* framework with multiple backend analysis tools, formalisms, solvers and frontend languages.

Precise Simulation of XSTS Models

We implemented a simulator for XSTS models in Theta, based on existing parts of the model-checking framework. The simulator can track the value of every variable and uses the transfer function of the model checker to calculate every successor state of the current state of the simulation. In the case of non-determinism (e.g., the selection from multiple fireable transitions), the simulator makes the choice explicit to the user so they can precisely control the simulation step-by-step. By extending Theta, we guarantee that the simulator executes the XSTS model in the exact way as Theta, inheriting its formal semantics.

[2] `https://github.com/ftsrg/theta`

Traversing Executions with the Simulator

In order to traverse every execution, we implemented an automated listener interface for the simulator, that follows a *depth first search* algorithm. At every decision point, where there are more successor states, it selects the first one. At the end of an execution, it tells the simulator to backtrack to the last decision point with unexplored successors, and repeats the algorithm, until every decision has been explored.

Graphical Representation

In order to visualize the resulting trace, we reused the visualization components of Theta, which can build generic graphs using the *dot* format of GraphViz[3]. We implemented a graph simplifier module, which can simplify the generated execution trace into a more readable, compact format by contracting common segments of the traces.

6.2 Evaluation Strategy

We used the following strategy during evaluation. We modeled a subset of the PSSM Test Suite in Gamma, then we generated their execution traces in various granularities. Finally, we compared the generated execution traces to the specified valid traces by pinpointing their differences:

- If the execution traces are *equivalent* (✓) to the ones specified by the standard, then our mapping is *correct*.
- If the generated execution traces are *missing* (−) executions from the specified valid traces, then the mapping is either incorrect, or the standard specifies incorrect executions.
- If the generated execution traces have *additional* (+) executions, then the mapping is either incorrect, or the standard is missing crucial execution information.

We formulated the following questions to evaluate the PSSM validation workflow.

Q1 – Can the PSSM validation workflow reproduce the traces given by the PSSM Test Suite?

Q2 – Can we gain deeper insights about the semantics from the detailed execution traces?

Q3 – Can the validation workflow expose previously unknown errors in the PSSM Test Suite?

6.3 Test Model Library

We manually created an extendable test model library in Gamma, containing all the common elements for Gamma test models as well as the Gamma version of a subset of the PSSM Test Cases. Tee PSSM Test Suite defines 18 test categories, with a total of 103 Test Cases. Since the Suite has an extensive feature set, we have selected specific tests and features that provide a good foundation for our validation.

[3] `https://graphviz.org/`

	Behavior	Transition	Event	Entering	Exiting	Entry	Exit	Choice	Junction	Fork	Join	Terminate	Final	History	Deferred	Redefinitio	Standalone	Other Test	**Total**
# Cases	5	15	16	5	5	6	3	5	5	2	3	3	1	8	10	6	3	1	**103**
# Models	5	9	11	3	3	0	0	5	0	1	2	0	1	4	0	0	0	0	**44**

Table 6.1: Number of modeled test cases from the PSSM Test Suite by category.

	Simple Completion Transition	Composite Completion Transition	Conflicting Transitions	Internal Transition	Local Transition	Guarded Transition	Operation Trigger	Event Deferring	DoActivity	Orthogonal Regions	Entry / Exit Point	Fork / Join State	Decision State	Junction State	Terminate State	History State
# Cases	76	33	4	3	5	16	7	10	10	34	19	8	6	8	3	8
# Models	32	11	4	2	0	5	0	0	5	11	0	1	5	0	0	4

Table 6.2: Models using specific feature set in the PSSM Test Suite. The feature sets are not disjunct, i.e., a single model can be in multiple columns in this table.

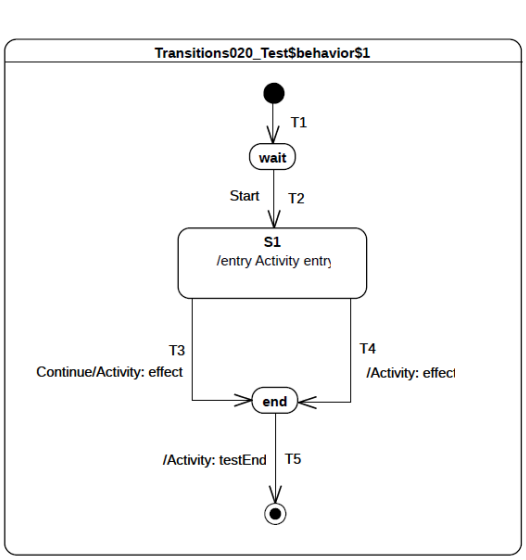
Table 6.1 shows the number of test cases PSSM defines in each category, and how many we have implemented of the specified models. Table 6.2 shows the number of models using a given feature set, and our models covering them. With this initial implementation, our aim was to cover the *most common* features and models, thus providing us with a baseline that can be further improved upon in future works.

We manually modeled each test case as separate Gamma model files grouped into folders specified by their categories. The Test Models contain the Message Queues of the test, the Target Statechart Component, and the connecting Test System.

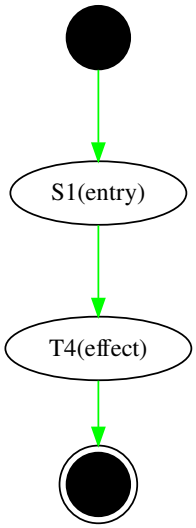
The transformation and trace generation has been automated as one-click processes, in order to make rapid-prototyping easier during model development; each test model has an associated Gamma generator file, which generates the detailed, split XSTS model. Each category also has a generator file calling their test cases. Finally, a main generator file calls all categories' generator files. The execution traces can also be generated automatically using a *batch simulator*, which processes all test models systematically, providing the visualized execution traces in various granularities.

6.4 Validation Results

Using the constructed models we generated the execution traces in two different granularities: a *baseline* model that contains only the original *tracing* information, and a *detailed* model injected with state *entry* and *exit* trace, transition *effect* trace, event raising trace, and RTC step trace information. Table 6.3 showcases the differences between the generated traces and the ones provided by the PSSM Test Suite. We used the same notation



(a) State machine of *Transition-020* model [12].



(b) The only execution trace of *Transition-020*, found by tracking the log variables.

Figure 6.1: State machine and execution trace of *Transition-020*.

introduced previously for the differences in the generated traces, extended with ✓⋆, which means the trace provides additional semantical information. Table 6.4 shows the aggregated results of the trace comparison.

6.4.1 Equivalent Traces

To answer our **Q1** question, we must compare the *Baseline* execution traces to the ones specified in the PSSM Suite. Indeed, we have generated 30 traces that match the ones specified in the specification precisely.

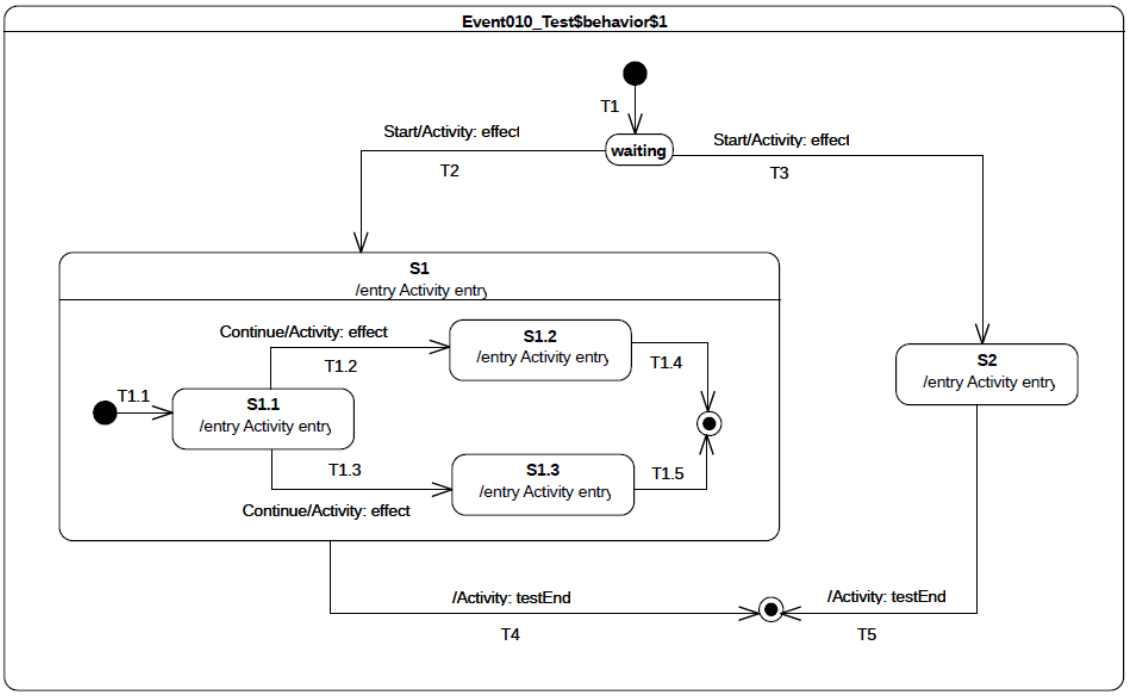
Let's take the *Transition-020* model as an example for equivalent trace. Figure 6.1a shows its State Machine. This test aims to showcase the priority of completion events over traditional ones. The execution trace generated by the validation workflow is presented in Figure 6.1b, which is the only valid execution trace specified by the Test Suite.

Another example could be the *Event-010* test case, which has three valid traces specified by the Test Suite. Figure 6.2a shows its State Machine. The execution trace generated by the validation workflow is shown in Figure 6.2b. It showcases how compactly it can display the three separate executions in a single diagram.

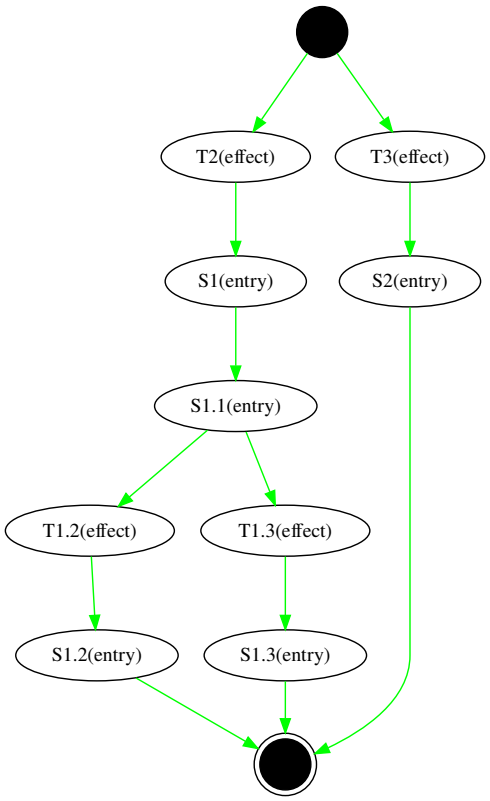
Using the generated detailed execution traces, we may also deduce more information than the Test Suite provides. Take RTC steps for example. In the PSSM Test Suite, only one RTC step is specified for each test case. Given, that there are several test models with more than one valid execution traces, we may lose important insight of the behavior of the test models. Also, there are several examples in the Test Suite, in which the RTC step specifies several transitions firing at once – but does not specify an ordering between them. As an answer to **Q3**: since the validation workflow may insert RTC beginning and RTC end trace information into the model, we can uncover the exact ordering all transitions in each model.

Test Case		Baseline	Detailed	PSSM Errors	Mapping Errors
Behavior	001	✓	✓⋆		
	002	✓	✓⋆		
	003-A	+	+	Ⓐ	
	003-B	+	+	Ⓐ	
	004	+	+	Ⓐ	
Transition	001	✓	✓⋆		
	007	✓	✓⋆		
	010	✓	✓⋆		
	015	✓	✓⋆		
	016	✓	✓⋆		
	017	−	−+	Ⓒ	Ⓘ
	019	−+	−+	Ⓒ	Ⓙ
	020	✓	✓⋆		
	022	✓	✓⋆		
Event	001	✓	✓⋆		
	002	✓	✓⋆		
	008	✓	✓⋆		
	009	✓	✓⋆		
	010	✓	✓⋆		
	015	✓	✓⋆		
	016-A		no result		
	016-B	✓	✓⋆		
	017-A	✓	✓⋆		
	017-B	✓	✓⋆		
	018	✓	✓⋆		
Entering	005	✓	✓⋆		
	010	−	−+	Ⓒ	Ⓘ
	011		no result		
Exiting	001	✓	+	Ⓒ	
	003	✓	+	Ⓒ	
	005	✓	+	Ⓒ	
Choice	001	✓	✓⋆		
	002	✓	✓⋆		
	003	✓	✓⋆		
	004	✓	✓⋆		
	005	✓	✓⋆		
Fork	002		no result		
Join	001	✓	✓⋆		Ⓙ
	002	−	−+	Ⓒ	Ⓙ
Final	001	✓	✓⋆		
History	001-A	−	−		Ⓗ
	001-B	−	−		Ⓘ
	001-C	−	−		Ⓗ
	001-D	−	−		Ⓗ

Table 6.3: Trace comparison results for the modeled test cases.



(a) State machine of *Event-010* model [12].



(b) The execution traces of *Event-010*, found by tracking the log variables.

Figure 6.2: State machine and execution trace of *Event-010*.

Test Case	Baseline	Detailed
# of equivalent traces	30	27
# of valid additional traces	4	10
# of valid missing traces	0	0
# of invalid additional traces	0	2
# of invalid missing traces	8	8
# of indeterminate traces	3	3

Table 6.4: Trace comparison results aggregated.

6.4.2 Different Traces

Several generated traces were different from the ones specified by the Test Suite. These models are denoted with either $+$, $-$ or $-+$. We compared the generated execution traces with the PSSM Specification, and aggregated the results for these traces. Table 6.3 contains a *PSSM Errors* and a *Mapping Errors* column, which contains the symbol of the found errors.

Unsynchronized behavior of doActivities Ⓐ The unsynchronization of the execution of doActivities has been presented in detail during the work. We first introduced it in Section 3.3.3, which we refined in Section 4.3 by giving an example interaction, and in Section 5.4 by actually showing the traces generated for such models.

Concurrency in orthogonal regions Ⓒ There are several tests checking the correct ordering of the effects on orthogonal regions. However, there were test cases, that did not specify their tracing in high enough detail, to provide all the possible orderings as valid execution traces.

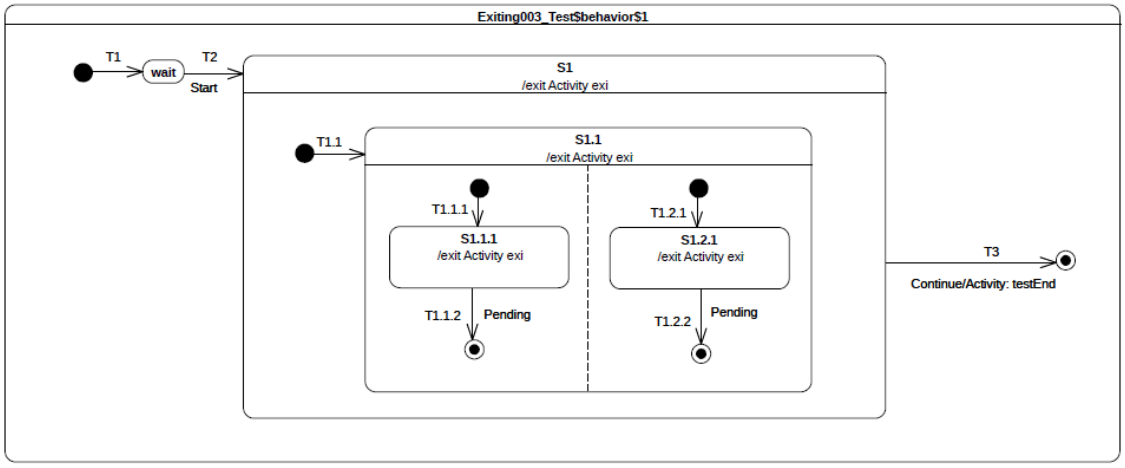
The *Exiting-003* model is shown in Figure 6.3a. The execution traces found by our workflow tracking the log variables are presented in Figure 6.3b.

Limitations in the Gamma mapping We found several limitations of the Gamma mapping during the validation workflow. Ⓗ Gamma did not transform transitions directed into History states in the same region as the source, and provided no grammar validations when trying to do so. Ⓙ Transitions directed into Join nodes may not have effects on them, preventing us from modeling several test cases. Ⓘ Gamma executes the initial transitions and entry actions for composite states in the wrong order: executing first all transition effects and then the entry actions.

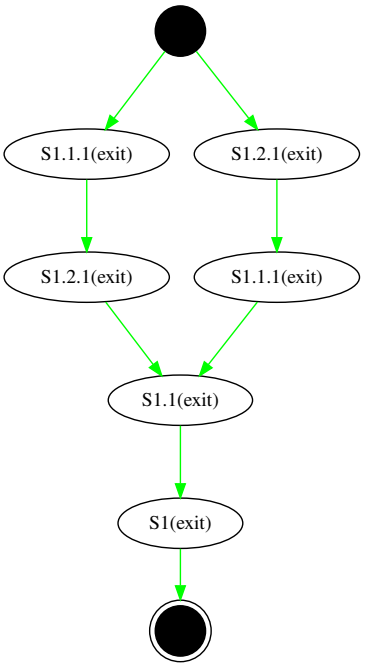
The maintainers of Gamma have been notified of these limitations.

6.4.3 Indeterminate Traces

There were three models in the test library, for which the trace generator could not provide meaningful traces. These models have been annotated with the “no result” line in Section 6.4.



(a) State machine of *Exiting-003* model [12].



(b) The execution traces of *Exiting-003*, found by tracking the log variables.

Figure 6.3: State machine and execution trace of *Exiting-003*.

6.5 Summary

Using the PSSM Validation Workflow, we were able to generate baseline and detailed execution traces for a subset of test cases using the essential features of UML State Machines. The execution traces were able to reproduce the execution specified by the Test Suite and provide deeper semantical insight into the underlying behavior (e.g., RTC steps). Using the validation workflow we were also able to reveal two types of errors in the test suite: the unsynchronized behavior of doActivities, and the concurrent nature of orthogonal regions, which were not covered in full detail by the test suite. The mapping has also shown previously unknown limitations in the Gamma language, which have been reported to the maintainers.

Chapter 7

Conclusion and Future Work

The preciseness and unambiguity of modeling langauge specifications are essential in the world of MBSE. However, the standardization process of such languages is a long and hard work. In this work, we proposed a novel validation workflow that aims to support the standardization process by harnessing the power of formal methods for generating detailed and semantically correct execution traces. The set of generated execution traces can be considered complete, i.e., it contains all possible execution traces for a model, due to the formal methods based approach. These execution traces provide valuable insight into the behavioral aspects of the modeling language.

The results of the work are twofold. From the theoretical point of view, we developed a mapping to model UML/PSSM State Machines in the Gamma formalism in a semantically sound way. In order to do so, we formalized a new Activity Component formalism, which allows the application of Gamma Activities as seperate components. We also defined a way to systematically generate precise execution traces from Gamma models in a configurable way, which results in highly detailed and compact execution graphs.

From the practical point of view, we extended the Gamma and Theta tools with new components neccessary for the modeling and execution trace generation of PSSM State Machines. As evaluation, we modeled a subset of the PSSM Test Suite models and did an extensive comparison over the generated execution traces and the ones specified in the standard. This evaluation demonstrated the applicability and the value of the approach, by showing that the validation workflow (i) can generate equivalent execution traces to the standard, and is also (ii) able to find certain executions the PSSM Test Suite did not show.

Future work The validation workflow seems to be a promising way to support the process of creating new standards and validating already existing ones. As direct next steps, we intend to implement an automatic UML $\rightarrow$ Gamma model transformation, in order to further simplify the validation of test models. Since the most prevalent error in the PSSM Test Suite execution traces seems to be missing concurrent traces, we also plan to formulate additional concurrent model features, such as Event Deferring in states, since they are extensively used with doActivities. Furthermore, we would like to extend the execution trace generation with fine grained granularity.

Chapter 8

Acknowledgement

We would like to express our gratitude to our advisors – Márton Elekes, Bence Graics, and Vince Molnár – for their continous support and guidance. We are also grateful for the valueable feedbacks of András Vörös and Benedek Horváth.

Bibliography

[1] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. A survey of modeling language specification techniques. *Information Systems*, 87:101425, 2020. ISSN 0306-4379. DOI: https://doi.org/10.1016/j.is.2019.101425. URL https://www.sciencedirect.com/science/article/pii/S0306437919303035.

[2] Manfred Broy and María Victoria Cengarle. UML formal semantics: lessons learned. *Software & Systems Modeling*, 10(4):441–446, Oct 2011. ISSN 1619-1374. DOI: 10.1007/s10270-011-0207-y. URL https://doi.org/10.1007/s10270-011-0207-y.

[3] Márton Elekes and Zoltán Micskei. Towards Testing the UML PSSM Test Suite. In *10th Latin-American Symposium on Dependable Computing, LADC 2021*, pages 1–4. IEEE, 2021. DOI: 10.1109/LADC53747.2021.9672570.

[4] Márton Elekes, Vince Molnár, and Zoltán Micskei. Assessing the specification of modelling language semantics: A study on UML PSSM, 2022. URL https://doi.org/10.21203/rs.3.rs-1577254/v1.

[5] Bence Graics. Mixed-Semantics Composition of Statecharts for the Model-Driven Design of Reactive Systems. Master's thesis, BME, 2018.

[6] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19(6):1483–1517, Nov 2020. ISSN 1619-1374. DOI: 10.1007/s10270-020-00806-5. URL https://doi.org/10.1007/s10270-020-00806-5.

[7] Object Management Group. Systems Modeling Language v2 (SysMLv2). URL https://github.com/Systems-Modeling/SysML-v2-Release.

[8] Object Management Group. Systems Modeling Language (SysML), 2012. URL https://www.omg.org/spec/SysML/1.6/About-SysML.

[9] Object Management Group. Systems Modeling Language (SysML) v2 RFP (ad/2017-12-02), 2017. URL http://doc.omg.org/ad/2017-12-2.

[10] Object Management Group. Unified Modeling Language (UML-v2.5.1), 2017. URL https://www.omg.org/spec/UML/2.5.1/About-UML.

[11] Object Management Group. Precise Semantics of UML Composite Structures (PSCS-v1.2), 2019. URL https://www.omg.org/spec/PSCS/1.2/About-PSCS.

[12] Object Management Group. Precise Semantics of UML State Machines (PSSM-v1.0), 2019. URL https://www.omg.org/spec/PSSM/1.0/About-PSSM.

[13] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML-v1.2), 2021. URL https://www.omg.org/spec/FUML/1.5/About-FUML.

[14] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, pages 1051–1091, Aug 2020. DOI: 10.1007/s10817-019-09535-x. URL https://doi.org/10.1007/s10817-019-09535-x.

[15] Lucas Lima, Alvaro Miyazawa, Ana Cavalcanti, Márcio Cornélio, Juliano Iyoda, Augusto Sampaio, Ralph Hains, Adrian Larkham, and Vaughan Lewis. An integrated semantics for reasoning about SysML design models using refinement. *Software & Systems Modeling*, 16(3):875–902, Jul 2017. ISSN 1619-1374. DOI: 10.1007/s10270-015-0492-y. URL https://doi.org/10.1007/s10270-015-0492-y.

[16] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A formal semantics for complete uml state machines with communications. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, pages 331–346, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38613-8.

[17] Shuang Liu, Yang Liu, Jun Sun, Manchun Zheng, Bimlesh Wadhwa, and Jin Song Dong. Usmmc: A self-contained model checker for uml state machines. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 623–626, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. DOI: 10.1145/2491411.2494595. URL https://doi.org/10.1145/2491411.2494595.

[18] Qin Ma, Monika Kaczmarek-Heß, and Sybren de Kinderen. Validation and verification in domain-specific modeling method engineering: an integrated life-cycle view. *Software and Systems Modeling*, Oct 2022. ISSN 1619-1374. DOI: 10.1007/s10270-022-01056-3. URL https://doi.org/10.1007/s10270-022-01056-3.

[19] Markus Maurer. *Automotive Systems Engineering: A Personal Perspective*, pages 17–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36455-6. DOI: 10.1007/978-3-642-36455-6_2. URL https://doi.org/10.1007/978-3-642-36455-6_2.

[20] 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 ICSE'18: Companion Proceeedings*, pages 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

[21] Milán Mondok. Formal verification of engineering models via extended symbolic transition systems. Bachelor's thesis, BME, 2020.

[22] Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. Model-Based Systems Engineering: An Emerging Approach for Modern Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):101–111, 2012. DOI: 10.1109/TSMCC.2011.2106495.

[23] G.J.M. Read, A. Naweed, and P.M. Salmon. Complexity on the rails: A systems-based approach to understanding safety management in rail transport. *Reliability Engineering & System Safety*, 188:352–365, 2019. ISSN 0951-8320. DOI: https://doi.org/10.1016/j.ress.2019.03.038. URL https://www.sciencedirect.com/science/article/pii/S0951832018311773.

[24] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003. DOI: 10.1109/MS.2003.1231147.

[25] Bran V. Selic. *On the Semantic Foundations of Standard UML 2.0*, pages 181–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-30080-9. DOI: 10.1007/978-3-540-30080-9_6. URL https://doi.org/10.1007/978-3-540-30080-9_6.

[26] Péter Szkupien and Vince Molnár. The effect of transition granularity in the model checking of reactive systems. In *Proceedings of the 29th Minisymposium of the Department of Measurement and Information Systems*, pages 54–57, 2022. DOI: 10.3311/MINISY2022-014.

[27] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of FMCAD'17*, page 176–179, Vienna, Austria, 2017. ISBN 978-0-9835678-7-5.

[28] Balázs Várady. Designing a Formally Verifiable Action Language for the Modeling of Reactive Embedded Systems. Bachelor's thesis, BME, 2019.

[29] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of cml: A formal modelling language for systems of systems. In *2012 7th International Conference on System of Systems Engineering (SoSE)*, pages 1–6, 2012. DOI: 10.1109/SYSoSE.2012.6384144.

[30] Ármin Zavada. Formal Modeling and Verification of Process Models in Component-based Reactive Systems. Bachelor's thesis, BME, 2021.

Appendix A

Splitting XSTS Transitions

In this section, we extend and formalize the *splitting* of XSTS transitions, originally introduced in [26]. In an XSTS model, the following operations cause internal non-determinism:

- *Choices*: The selection between the branches op_i of a choice op_1 or ... or op_n is non-deterministic.
- *Conditionals*: Although the evaluation of the condition ψ of a conditional (ψ) ? op_{then} : op_{else} is deterministic, the splitting of ψ, op_{then} and op_{else} makes the observation of the actual execution easier.
- *Parallels*: In every step of a parallel action op_1 || ... || op_n, the selection between the not-finished branches op_i is non-deterministic.
- *Havocs*: The concrete value $x \in D_v$ assigned to variable $v \in V$ is non-deterministically selected from domain D_v of variable v. This non-determinism can not be made external, but havocs are for modeling the non-deterministic inputs from the environment. In the scope of PSSM models, there is no non-deterministic input, so we just ignore havocs in this work.

The splitting of an XSTS model should not change the possible executions. Informally, it just replaces some non-deterministic operations with deterministic ones and transforms the original internally non-deterministic semantics into external non-determinism by moving every non-deterministic step outside the split transitions.

In other words, the goal of splitting is to eliminate the abstract states from the execution of an XSTS model, making the successor state s' of every transition $t = (s, s')$ concrete: $|s'| \leq 1$.

In general, splitting is a *model-transformation* over an XSTS model $\langle V, Tr, In, En \rangle$ which breaks down some monolithic transitions of the transition sets Tr, In, En into smaller transitions (called $fragments$), resulting in a split XSTS model $\langle V', Tr', In', En' \rangle$.

The splitting of a transition relation T means the splitting of every transition $t \in T$. Note, that for each transition, splitting yields at least one split transition, so the split transition set T' must not contain fewer transitions than the original T: $|T'| \geq |T|$.

A.1 Splitting Rules

In the following, we formalize the splitting of a transition relation T by defining splitting rules of *choices*, *conditionals*, and *parallels*. In order to make the originally internal non-deterministic choices external, we introduce a new variable pc which will serve as a program counter, to enforce the original control flow: $V' = V \cup \{pc\}$, $D_{pc} = integer$, $IV(pc) = 0$.

The splitting of a transition $t \in T$ results in a set of split transitions (fragments): $split(t) = \{t'_1, \ldots, t'_n\}$. The splitting of a transition relation T results in the union of the fragments of every original transition $t \in T$: $split(T) = \bigcup_{t \in T} split(t)$.

A fragment of an operation op wraps the operation into a sequence, starting with an assumption on pc and ending with an assignment to pc. Formally, $frag(op, x, y) = ([pc = x], op, pc := y)$, where x is the program counter value, after which op can execute, and y is the program counter value associated with the fragment. These assumptions and assignments will guarantee the original control flow of the model.

Note, that the first fragment(s) of the original non-split transition should start with $[pc = 0]$, while the last fragment(s) should end with $pc := 0$. This means, that the original states of the system (i.e., where the system is not in the middle of the execution of an original atomic transition) are the states, where $pc = 0$ holds.

In the following, we use $split(op, x, y)$, where x denotes the pc value which should be assumed at the beginning of the first fragment(s) of op, and y denotes the pc value which should be assigned to pc at the end of the last segment(s) of op. For transitions $t \in T$, this means $split(t) = split(t, 0, 0)$.

The splittable operations are choices, conditionals, and parallels. In case of a sequence $seq = op_1, \ldots, op_n$, if op_i is the first splittable operation, the resulting fragments will be $split(seq, x, y) = \{(frag((op_1, \ldots, op_{i-1}), x, \xi_1), split(op_i, \xi_1, \xi_2), split((op_{i+1}, \ldots, op_n), \xi_2, y)\}$, where ξ_i denotes a unique program counter value, which can be generated incrementally, for example. If there is no splittable operation in seq, $split(seq) = frag(seq, x, y)$.

Example 1 (Splitting sequence with non-splittable operations). *For transition $t = ([x = 0], x := 1, y := 2)$ with no splittable operation, splitting will result in only a single fragment:*

$$split(t) = split(t, 0, 0) = frag(t, 0, 0) = \{([pc = 0], [x = 0], x := 1, y := 2, pc := 0)\}$$

The splitting of a *choice* of form $ch = (op_1 \text{ or } \ldots \text{ or } op_n)$ means splitting all of its branches op_i into fragments, with the same assumption, and the same assignment on pc. This will result in a set of fragments for each branch, from which exactly one non-deterministically selected set will execute. Formally, $split(ch, x, y) = \bigcup_{i=0}^{n} split(op_i, x, y)$.

Example 2 (Splitting choice). *For transition $t = (x := 1 \text{ or } x := 2)$ with a splittable choice with 2 branches, splitting will result in 2 fragments:*

$$split(t) = split(t, 0, 0) = \left\{ \begin{array}{l} ([pc = 0], x := 1, pc := 0), \\ ([pc = 0], x := 2, pc := 0) \end{array} \right\}$$

Example 3 (Splitting sequence with splittable and non-splittable operations). *For transition* $t = (y := x, (x := 1 \text{ or } x := 2), z := x)$ *with a splittable choice with 2 branches, in the middle of a sequence, splitting will result in 4 fragments:*

$$split(t) = split(t, 0, 0) = \begin{Bmatrix} ([pc = 0], y := x, pc := 1), \\ ([pc = 1], x := 1, pc := 2), \\ ([pc = 1], x := 2, pc := 2), \\ ([pc = 2], z := x, pc := 0) \end{Bmatrix}$$

The splitting of a *conditional* of form $cond = (\psi)\ ?\ op_{then}\ :\ op_{else}$ means splitting the condition into a separate fragment, as well as the splitting of op_{then} and op_{else}. In order to keep the original control flow, we need two pc values ξ_{then} and ξ_{else} for op_{then} and op_{else}, respectively. Formally, $split(cond, x, y) = \{condfrag(\psi, x, \xi_{then}, \xi_{else})\} \cup split(op_{then}, \xi_{then}, y) \cup split(op_{else}, \xi_{else}, y)$.

The condition fragment $condfrag(\psi, x, \xi_{then}, \xi_{else})$ checks $pc = x$, then assigns ξ_{then} or ξ_{else} to pc based on the evaluation of ψ, repsectively. Formally, $condfrag(\psi, x, \xi_{then}, \xi_{else}) = ([pc = x], pc := (\psi)\ ?\ \xi_{then}\ :\ \xi_{else})$, where an assignemnt of form $v := \psi\ ?\ a\ :\ b$ means evaluating the Boolean expression ψ, and assigning a to v, if ψ is true, or b, otherwise.

Example 4 (Splitting conditional). *For transition* $t = ((x > 0)\ ?\ y := x\ :\ y := 0)$ *with a splittable conditional, splitting will result in 3 fragments:*

$$split(t) = split(t, 0, 0) = \begin{Bmatrix} ([pc = 0], pc := (x > 0)\ ?\ 1\ :\ 2), \\ ([pc = 1], y := x, pc := 0), \\ ([pc = 2], y := 0, pc := 0) \end{Bmatrix}$$

The splitting of a *parallel* of form $par = op_1 \mid\mid \ldots \mid\mid op_n$ means splitting every operation of every branch into a separate fragment, as well as creating a fragment for $forking$ and $joining$ the branches. For every branch op_i, a separate *branch program counter* pc_i is introduced, in order to guarantee the execution order of operations from one branch: $V' = V \cup \{pc_1, \ldots, pc_n\}$. The assumption on pc_i can be merged into the original pc assumption(s) at the beginning of the fragment with logical *and*.

In order to keep the original control flow between $fork$, branches, and $join$, a new pc value ξ is needed. Formally, $split(par, x, y) = \{forkfrag(x, \xi, \bigcup_{i=1}^{n} pc_i), \bigcup_{i=1}^{n} \bigcup_{j=1}^{|op_i|} parfrag(\xi, pc_i, op_i, j), joinfrag(\xi, y, \bigcup_{i=1}^{n} pc_i)\}$.

The $fork$ fragment $forkfrag(x, \xi, PC)$ checks $pc = x$, then assigns 1 to every branch program counter $pc_i \in PC$, and ξ to pc. Informally, the $fork$ fragment enables the execution of the parallel branches. Formally, $forkfrag(x, \xi, PC) = ([pc = x], seq_{i=1}^{|PC|} PC_i := 1, pc := \xi)$, where $seq_{i=1}^{n} op_i$ means the sequence of $op_1, \ldots, op_n$.

Generally, the jth operation of branch op_i results in parallel fragments $parfrag(\xi, pc_i, op_i, j) = \bigcup f'$. First, we split op_{i_j} into fragments with $split'(op_{i_j}, \xi, \xi)$ which will result in the fragments f of op_{i_j}. Then, we wrap each of these fragments f into a parallel fragment f', with adding an assumption $[pc_i = j]$ to the beginning, and an assignment $pc_i := \varphi$ to the end, where $\varphi = j + 1$, if $j < |op_i|$, otherwise 0. As a result, $pc_i = 0$ denotes, that the execution of op_i has finished.

We used $split'$ instead of $split$, because in order to enable every valid parallel execution, we need to split every operation op_i of sequences $op_1, \ldots, op_n$ into a separate fragment. So

$split'(op, x, y)$ only differs from $split(op, x, y)$ in the case of sequences, creating a separate fragment of every contained operation.

The *join* fragment $joinfrag(\xi, y, PC)$ checks $pc = \xi$, and $pc_i = 0$ for every $pc_i \in PC$, then assigns y to pc. Informally, the *join* fragment awaits the finishing of every parallel branch. Formally, $joinfrag(\xi, y, PC) = ([pc = \xi \wedge \bigwedge_{i=1}^{|PC|} pc_i = 0], pc := y)$.

Example 5 (Splitting parallel). *For transition* $t = ((x := 1, y := x) \mathbin{//} (x := 2, y := x))$ *with a splittable parallel with two 2-long sequences, splitting will use 2 branch program counters* pc_1, pc_2*, and result in 6 fragments:*

$$split(t) = split(t, 0, 0) = \left\{ \begin{array}{l} ([pc = 0], pc_1 := 1, pc_2 := 1, pc := 2), \\ ([pc = 1 \wedge pc_1 = 1], x := 1, pc_1 := 2), \\ ([pc = 1 \wedge pc_1 = 2], y := x, pc_1 := 0), \\ ([pc = 1 \wedge pc_2 = 1], x := 2, pc_2 := 2), \\ ([pc = 1 \wedge pc_2 = 2], y := x, pc_2 := 0), \\ ([pc = 1 \wedge pc_1 = 0 \wedge pc_2 = 0], pc := 0) \end{array} \right\}$$

A.2 Merging Transition Relations

Splitting every transition relation In, En, Tr of an XSTS model independently, may modify the original semantics of the model, because the execution order of the transition relations $In, En, Tr, En, Tr, \ldots, En, Tr$ would execute only fragments in this order, instead of originally atomic transitions.

To avoid this difference in the semantics of the non-split and split models, we merge every fragment $t \in split(In) \cup split(En)$ into $split(Tr)$, and force the original execution order of transition relations with explicit assumptions and assignments of newly introduced variables.

Formally, we extend the variables V of the system with two Boolean variables *init* and *trans*: $V' = V \cup \{init, trans\}$, $D_{init} = D_{trans} = bool$, $IV(init) = \top$, $IV(trans) = \bot$.

Informally, the value of *init* and *trans* denote which original transition relation should execute:

- *init* denotes the execution of the original In transition relation
- $\neg init \wedge \neg trans$ denotes the execution of the original En transition relation
- $\neg init \wedge trans$ denotes the execution of the original Tr transition relation

In order to enforce these rules, we extend every fragment $t = op$, $op \in Ops$ with the following operations, resulting in t':

- $t \rightsquigarrow t' = ([init], op, init := \bot)$ for every t which is created from a $t_{In} \in In$
- $t \rightsquigarrow t' = ([\neg init \wedge \neg trans], op, trans := \top)$ for every t which is created from a $t_{En} \in En$
- $t \rightsquigarrow t' = ([\neg init \wedge trans], op, trans := \bot)$ for every t which is created from a $t_{Tr} \in Tr$

After these transformations, we can merge the t' fragments from $split(In)$, $split(En)$, and $split(Tr)$ into Tr', while $In' = En' = \emptyset$. The resulting $\langle V', Tr', In', En' \rangle$ model will have the same executions as the original model has.

Example 6 (Merging transition relations). *Given an XSTS model $\langle V, Tr, In, En \rangle$, where $Tr = \{tr_1, tr_2\}$, $In = \{in_1, in_2\}$, and $En = \{en_1, en_2\}$ the merged XSTS model will be $\langle V', Tr', In', En' \rangle$, where $V' = V \cup \{init, trans\}$, $In' = En' = \emptyset$, and*

$$Tr' = \left\{ \begin{array}{llll} (& [init], & in_1, & init := \bot \ ), \\ (& [init], & in_2, & init := \bot \ ), \\ (& [\neg init \wedge \neg trans], & en_1, & trans := \top \ ), \\ (& [\neg init \wedge \neg trans], & en_2, & trans := \top \ ), \\ (& [\neg init \wedge trans], & tr_1, & trans := \bot \ ), \\ (& [\neg init \wedge trans], & tr_2, & trans := \bot \ ) \end{array} \right\}$$