MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Szkupien Péter
Mérnökinformatikus hallgató részére

Komponensalapú reaktív rendszerek lépésenként vezérelhető szimulációja precíz formális szemantika szerint

Kritikus rendszerek analízisekor gyakran használunk precíz, jellemzően reaktív komponensekből építkező modelleket a szándékolt viselkedés leírására, melyeket szimulációval, teszteléssel, esetleg formális módszerekkel vizsgálhatunk. A szimuláció célja sokféle lehet: a modell „kipróbálása", „in-the-loop" módszerek, vagy a valódi rendszerben megfigyelt, esetleg a formális ellenőrzés által visszaadott hibás lefutások elemzése.

A legtöbb szimulációs eszköz megelégszik egy-egy reprezentatív lefutás generálásával. Ennek oka, hogy a magas szintű mérnöki modellek szemantikája általában nem elég precíz, illetve sok szimulátor fix vagy randomizált választásokkal kezeli le a belső nemdeterminisztikus viselkedéseket – ezek nem a környezet (itt a felhasználó) döntése miatt nem-determinisztikusak, hanem a végrehajtási szemantika vagy explicit modellezés miatt.

Ugyanakkor ez azt is jelenti, hogy az ilyen szimulátorok alkalmatlanok tetszőleges lefutás visszajátszására, ami nélkül nem lehet vizsgálni a megfigyelt vagy formális módszerekkel talált lefutásokat. Szükség van tehát olyan szimulátorokra, amelyek a modellek precíz formális szemantikája szerint, lépésről lépésre, a belső nemdeterminizmust is kívülről vezérehetővé téve képesek szimulációt végezni.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a komponensalapú reaktív rendszerek kontextusában egy kiválasztott környezetben (modellezési nyelv és eszközkészlet), hogy milyen nemdeterminisztikus viselkedések jellemzőek egy modellben és milyen problémákat jelent ezek kezelése.
- Javasoljon olyan megoldást, amellyel a belső nemdeterminisztikus döntések vezérelhetősége a modell formális szemantikája szerinti összes lehetőséget támogatva megvalósulhat.
- Tervezzen meg egy szimulációs keretrendszert, amely a javasolt módszer segítségével lehetővé teszi magas szintű modellek precíz szimulációját.
- Implementálja a keretrendszert, majd esettanulmányokkal demonstrálja a módszer alkalmazhatóságát.

Tanszéki konzulens: Dr. Molnár Vince, adjunktus

Budapest, 2022.03.19.

……..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Step-by-Step Controllable Simulation of Component-Based Reactive Systems Based on Precise Formal Semantics

Master's Thesis

Author
Péter Szkupien

Advisor
dr. Vince Molnár

December 18, 2022

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2022. december 18.

Szkupien Péter
hallgató

Contents

Kivonat

A szoftveres megoldások folyamatosan terjednek az élet minden területén, amivel a biztonságkritikus szoftverrendszerek is egyre bonyolultabbá válnak. Ezen rendszerek minden körülmények között helyesen kell, hogy működjenek, ellenkező esetben a hibás működés katasztrofális következményekkel járna. Biztonságosságuk garantálására új fejlesztési elvek és módszerek terjedtek el, többek között a modellalapú rendszertervezés.

A modellalapú rendszertervezés alkalmazásához a mérnököknek magasszintű modellezési nyelvekre van szükségük, amelyek segítségével leírhatják a rendszerek különböző aspektusait. A rendszerek alapvető aspektusa a dinamikus viselkedés, amely leírására egyre népszerűbb a végrehajtható modellek használata. Ezek a modellek precíz formális szemantikával rendelkeznek, amelyet alacsonyszintű matematikai fogalmak segítségével definiálnak. A mérnökök által használt magasszintű modellek, és alacsony szinten definiált szemantikájuk közti absztrakciós szakadék megnehezíti a magasszintű modellek szemantikájának precíz megértését. Gyakori példa erre a magasszintű modellek atomi lépéseiben lévő belső nemdeterminizmus, amely csak a mögöttes alacsony szinten jelenik meg. Általános igény a mérnökök részéről, hogy a modelljeik szimulációja segítségével megfigyelhessék azok pontos viselkedését, ám a belső nemdeterminizmus a végrehajtások bizonyos részeit vezérelhetetlenné és megfigyelhetetlenné teszi, csökkentve ezzel a szimuláció precizitását.

Diplomatervemben bemutatok különböző absztrakciós szintű modellezési nyelveket, amelyeket komponensalapú reaktív rendszerek modellezésére használnak. A szemantikájukat alapul véve azonosítom a végrehajtásaik során előforduló lehetséges nemdeterminisztikus döntéseket, és bemutatom a saját algoritmusomat, amely a belső nemdeterminizmusokat külsővé alakítja. Erre építve megtervezek egy szimulációs keretrendszert, amely lehetővé teszi a felhasználó számára a szimuláció lépésenkénti vezérlését azáltal, hogy minden nemdeterminisztikus lépés vezérlését a felhasználóra bízza. Bemutatom a szimulátor különböző felhasználási módjait, amelyekhez megfelelő vezérlési mechanizmusokat is definiálok.

Az elméleti eredményeimet nyílt forráskódú eszközök (Gamma állapotgép-kompozíciós keretrendszer, Theta modellellenőrző keretrendszer) kiterjesztéseként implementáltam, amit tömören szintén bemutatok. Végül egy esettanulmányon keresztül elemzem a munkám eredményét, és demonstrálom annak használhatóságát.

Abstract

Safety-critical software systems are becoming more and more complex with the continuous spreading of software solutions in all areas of life. These systems must operate correctly under all circumstances, otherwise, their fault would cause catastrophic consequences. Therefore, for guaranteeing safety, new development principles have been introduced, such as model-based systems engineering (MBSE).

To apply MBSE, engineers need high-level modeling languages for describing the different aspects of systems. One of the crucial aspects is the dynamic behavior, for which the application of executable models is spreading. These models have their precise formal semantics defined by low-level mathematical concepts. The abstraction gap between the high-level models used by engineers, and their semantics defined on a low level, makes it harder to understand the precise semantics of a high-level model. A common example of this is the internal non-determinism inside high-level atomic steps, which may only occur in the hidden low-level. It is a general need for engineers to be able to simulate their models, in order to observe their behavior precisely. Internal non-determinism makes some parts of the executions uncontrollable and unobservable, causing imprecise simulation.

In this thesis, I present some modeling languages with different abstraction levels which are used to model component-based reactive systems. Based on their semantics, I analyze the possible non-deterministic decisions of their execution and propose my own algorithm to make internal non-determinism external. Building on this, I design a simulation framework, which enables the user to control the simulation step-by-step, by making every non-deterministic decision explicit. I also identify different use cases, for which different simulation control mechanisms are presented.

I implemented my theoretical results as parts of open-source tools – the Gamma Statechart Composition Framework, and the Theta Model Checking Framework – which I also briefly present. Finally, the use of my work is discussed through a case study.

Chapter 1

Introduction

With the continuous spreading of software solutions in all areas of life, software systems are becoming more and more complex. The increasing complexity requires new approaches to guarantee software quality, by using higher abstraction levels during design to reduce complexity. For this, engineers need convenient high-level modeling languages.

Models play a central role in the development process, especially in safety-critical domains (e.g., automotive, railway, aerospace industry) where engineering companies must use specific approaches in order to meet the corresponding standards. Model-based systems engineering (MBSE) is an actively spreading methodology for the development of complex, safety-critical systems [14].

In MBSE, models are used to describe several different aspects of the designed system. A fundamental aspect is the behavior of the system, for which many formalisms are available for modeling. Due to the fact that complex systems usually have complex behavior, a high abstraction level is necessary for this aspect, too.

It is a common need for engineers to be able to observe the behavior of the designed system in design time to receive early feedback – saving both working hours and money. To achieve this goal, a simulator for behavior models is necessary.

In order to precisely simulate behavioral models, having their formal semantics defined is crucial. Due to the high abstraction level of modeling languages used by engineers, the complexity of semantics is also high, making the simulation of such models challenging.

A common problem with complex semantics is non-determinism. Inside a single step of the high-level model, there may be non-deterministic decision points, enabling multiple different executions of a single step. Unfortunately, these inner decision points may remain hidden from the modeler, because executions with only inner differences can still lead the system to the same observable state. Furthermore, the different outputs can not be explained without the observation of the inner decisions.

In order to give full control to the engineers to be able to traverse any legal execution they want, the internal non-determinism of high-level atomic steps should be revealed and made explicit to the user. This could be achieved by breaking down the internally non-deterministic steps into smaller deterministic parts without changing the original semantics of the model.

In this thesis, I present the high-level statechart formalism for the design of component-based reactive systems, in which I present some examples of non-deterministic behavior. Statechart models can be transformed, e.g., into the lower-level extended symbolic

transition system (XSTS) formalism [15], where non-determinism can still occur. I systematically analyze every source non-determinism in the XSTS formalism and propose an algorithm (called splitting) to make the internal non-determinism inside transitions external, enabling their observation.

Building on the splitting algorithm, I design a step-by-step controllable simulation framework for the precise simulation of split XSTS models. For guaranteed adherence to semantics, the simulation framework relies on an existing model checking infrastructure which is used for the semantics-critical calculation of successor states during the simulation.

This simulator enables the user to explicitly control every decision point during the simulation, regardless of whether it was originally internal or external. Based on the identified use cases, I also define and implement different control mechanisms for the simulator.

The rest of this thesis is structured as follows. In Chapter 2, I give an overview of the theoretical background behind my work: formal methods, formal models, simulation, and the frameworks I contributed to. In Chapter 3, I define requirements on the simulation framework and design its high-level architecture. In Chapter 4, I formally present my XSTS transition splitting algorithm to make internal non-determinism external. In Chapter 5, I detail the simulation framework focusing on how it satisfies the requirements. In Chapter 6, I present the use of the splitting algorithm and the simulation framework through a case study. Lastly, in Chapter 7, I summarize my work and present some future work.

Chapter 2

Background

In this chapter, I summarize the related background to this thesis. I briefly present formal methods and model checking (Section 2.1), and formal models of different abstraction levels (Section 2.2). I also present the need for simulation (Section 2.3) and the tools related to my work (Section 2.4).

2.1 Formal Methods

Formal methods are precise approaches for the analysis of systems. They rely on mathematical logic, therefore they can be used only on mathematically precise *formal models*. They are widely used for verification purposes in safety-critical domains (e.g., automotive, railway, and aerospace industries) where it is crucial to guarantee the safety of systems.

2.1.1 Model Checking

Model checking [4] is a formal method for the algorithmic analysis of the dynamic behavior of systems, checking whether a system satisfies a given requirement or not. The high-level overview of model checking is shown in Figure 2.1.

To apply model checking, three building blocks are required [3]:

- *Formal model*: A mathematically precise model of the behavior of the system is required, usually a *transition system* [3].
- *Formal requirement*: The verifiable requirement should also be in a mathematically precise form, for which *temporal logic* [17] is appropriate.
- *Algorithm*: Finally, an (effective) algorithm is required to decide whether the given formal model satisfies the given formal requirement or not. The output of the algorithm is a *proof* or a *counterexample*.

Although with traditional *testing* bugs can be found in systems, the correctness of a system can not be *proved*. Model checking explores the entire state space of the system, making it possible to mathematically *prove* that the system satisfies a certain requirement.

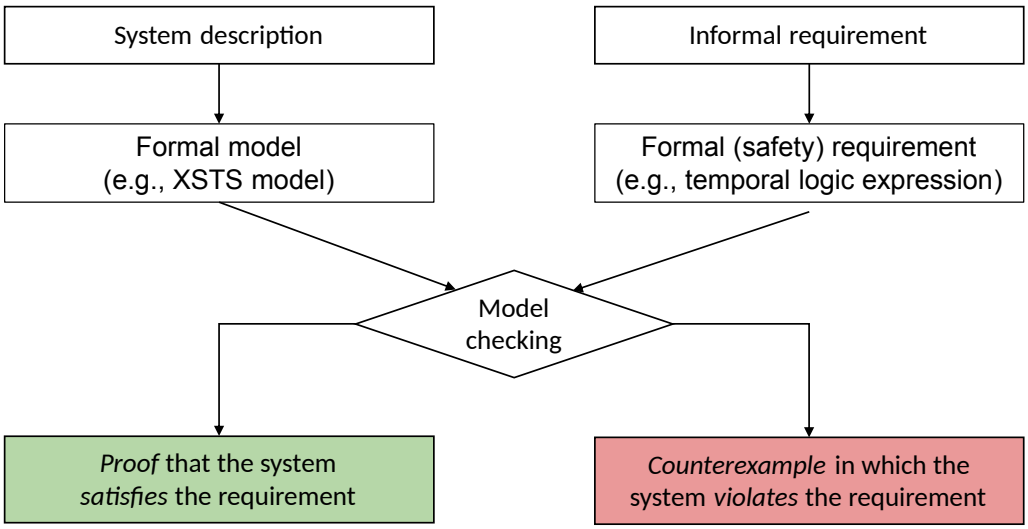

Figure 2.1: Model checking.

2.2 Formal Models

As systems – especially safety-critical systems – are becoming more complex, new development approaches, such as *model-based systems engineering* (MBSE) [18], have been introduced to handle complexity. In MBSE, models are the primary artifacts of the development process. [19] To create these models, suitable modeling languages are required to model the different aspects of systems, e.g., structure or behavior.

In the case of *behavioral models*, it is especially important to have precise formal semantics in order to be able to *execute* them – either by directly simulating them or by automatically transforming them into an executable formalism, e.g., program code.

2.2.1 Component-Based Reactive Systems

Component-based development is a practical approach in the case of complex systems: engineers can compose such systems from reusable parts (*components*) which are communicating with each other through well-defined interfaces.

Reactive systems are interacting with their environment. These interactions are modeled as input and output events: input events coming from the environment can trigger internal behaviors (e.g., state change) of the system (i.e., the system *reacts* to an event), while the system can react with output events to its environment. The possible interactions between the system and its environment can also be modeled with interfaces.

In the case of *component-based reactive systems*, the components of the system can be modeled as *state machines* [15]. Informally, a state machine is a reactive component, because its behavior is modeled as *transitions*, triggered by input events and generating output events.

State machines are state-based models, focusing on the internal state of the system, and the transitions changing it. This modeling approach can be used at different abstraction levels. *Finite state machine* (Section 2.2.2) is a low-level mathematical, while *Statechart* (Section 2.2.3) is a high-level engineering formalism.

2.2.2 Mathematical Finite State Machine

Finite state machine is a mathematical formalism to model the state-based behavior of a system. Informally, it defines the possible states and state changes (*transitions*) of a system. It can be either *deterministic* or *non-deterministic*, according to the number of possible transitions at a specific state, triggered by a specific input event.

Definition 1 (Finite State Machine). Formally, a deterministic finite state machine is a 5-tuple $SM = \langle S, s_0, I, O, T \rangle$ where:

- $S = \{s_1, s_2, \ldots, s_n\}$ is a finite set of *states*.
- $s_0 \in S$ is the *initial state*.
- $I = \{i_1, i_2, \ldots, i_m\}$ is a finite set of *input events* coming from the environment.
- $O = \{o_1, o_2, \ldots, o_l\}$ is a finite set of *output events* going to the environment. Note, that I and O are disjoint, i.e., $I \cap O = \emptyset$.
- $T : (I \times S) \mapsto (S \times O)$ is the (fully defined) *transition function*, representing the reaction of the state machine to a specific input event in a specific state, i.e., the new state and the generated output event. ▪

In Definition 1, I define a fully-defined deterministic finite state machine, i.e., which can take exactly one transition at every state, reacting to every input event. Note, that this does not mean, that the system must always change its state because a transition can lead the system to the same state as it was already in. The lack of the generated output event can be modeled with a special output event ε, i.e., a transition $T(i, s) = (s, \varepsilon)$ means that the system does not react to input event i in state s.

Note, that the determinism of the state machine only means that the state machine itself is deterministic. Nevertheless, it can be used to model a system with a non-deterministic environment, by non-deterministically choosing the input events of the state machine.

Example 1 (Finite State Machine). *Consider a machine with two buttons,* do *and* reset*, which beeps after every third press of button* do*, while the press of button* reset *resets the counter. This machine can be modeled with the following deterministic finite state machine* $SM = \langle S, s_0, I, O, T \rangle$*:*

- $S = \{s_0, s_1, s_2\}$ *where* s_n *represents the state where* n do *presses are counted since the last* beep *or* reset *press.*
- $s_0 \in S$ *is the initial state.*
- $I = \{do, reset\}$ *where the button presses are modeled as input events.*
- $O = \{beep, \varepsilon\}$ *where beep models the beep of the machine as an output events, and* ε *models the lack of beep.*
- $T = \left\{ \begin{array}{lllll} (\ do, & s_0, & s_1, & \varepsilon &), \\ (\ do, & s_1, & s_2, & \varepsilon &), \\ (\ do, & s_2, & s_0, & beep &), \\ (\ reset, & s_0, & s_0, & \varepsilon &), \\ (\ reset, & s_1, & s_0, & \varepsilon &), \\ (\ reset, & s_2, & s_0, & \varepsilon &) \end{array} \right\}$

The transition function can be represented as a state-transition table. *Each cell represents the effect of the given input event in the given state as a pair of the next state and the output event.*

State/Input	s_0	s_1	s_2
do	(s_1, ε)	(s_2, ε)	$(s_0, beep)$
reset	(s_0, ε)	(s_0, ε)	(s_0, ε)

The state machine can also be represented graphically as a graph. The nodes represent the states, while the edges represent the transitions. An edge with label i/o represents a transition leading the system from the state represented by the source node to the state represented by the target node, triggered by input event i, producing output event o.

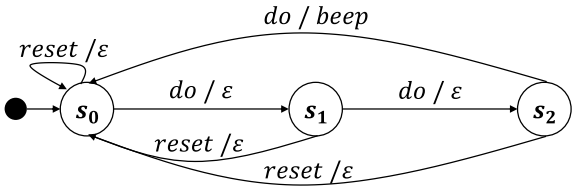

Figure 2.2: Graphical representation of the example finite state machine.

2.2.3 High-Level Statechart

The *statechart* formalism is a popular tool to model the behavior of state-based reactive systems in a higher abstraction level [11]. It relies on the mathematical *finite state machine* formalism but it is more expressive, making the formalism usable in everyday engineering work.

In general, the statechart formalism extends the finite state machine formalism with the following:

- *Hierarchically nested states*: The internal behavior of a state (*superstate*) can be modeled with an inner statechart (*substates*). The inner statechart operates in the context of the superstate. When the superstate is activated, its initial inner state activates, too. When the superstate is deactivated, its whole inner statechart deactivates.
- *Orthogonal regions*: The internal behavior of a state can be modeled with several statecharts (*regions*), running independently.
- *Entry and exit actions*: Like in the case of transitions, *entry* and *exit* actions can be defined for states. These actions are triggered when the state is activated or deactivated, respectively.

Example 2 (Statechart). *Consider the extension of the finite state machine defined in Example 1. A new input event* switch *is introduced to switch the whole machine on/off. When the machine is switched on/off, it beeps. When the machine is turned on, it beeps when the reset button is pressed first since the machine is turned on.*

It serves as an example for all of the three new features of the statechart formalism. The initial Off *state defines an entry and exit action for beeping. The hierarchical* On *state contains two orthogonal regions which are independently responsible for counting the* do *and* reset *events.*

The graphical representation of this statechart is shown below.

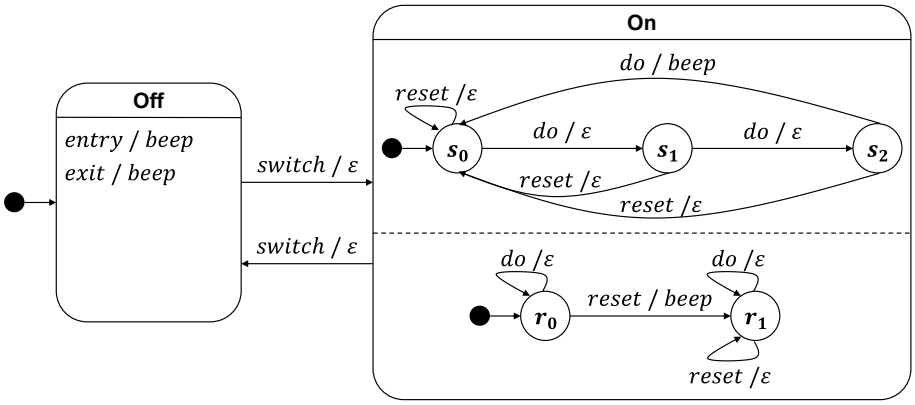

Figure 2.3: Graphical representation of the example statechart.

The statechart formalism can be further extended, e.g. with *variables* and *guards*. Guards on transitions are logical expressions over variables, and a transition can not be fired if its guard evaluates to false.

The informally introduced high-level formalism has several concrete definitions in different modeling languages such as the general-purpose *Unified Modeling Language*(UML) [7] and *Systems Modeling Language* (SysML) [9] developed by the *Object Management Group* (OMG), though these formalisms often lack precise formal semantics [5].

2.2.3.1 Non-Determinism

The execution of a statechart can be non-deterministic due to several reasons. In general, we distinguish two types of non-determinism: *internal* and *external*.

Internal non-determinism. A single step of a statechart can contain non-deterministic decision point(s). This means, that in a specific state configuration, with a specific input event, the statechart can behave in different ways. Due to the fact, that the environment can only interact with the statechart with input events (i.e., the statechart itself is a *black box* in terms of control), internal non-determinism makes the behavior of the system uncontrollable, causing problems in several use cases, e.g., precise simulation.

The systematic analysis of (internal) non-determinism in statechart semantics is beyond the scope of this work, here I only highlight some examples to underline its necessity.

- *Multiple fireable transitions*: In real-life engineering work, especially with the growing complexity of state hierarchies and guard expressions, statecharts are usually not deterministic and fully defined. This means, that in a specific state configuration, an input event may trigger multiple transitions. Although state machine semantics usually define some rules to select the transition to fire, in some cases neither of these rules defines the *exact* transition(s) to fire, i.e., the choice from the firable transitions remains non-deterministic.
- *Orthogonal regions*: The execution of orthogonal regions is independent, meaning that there is no guarantee on the order of their steps. As a result, the exact ordering is non-deterministic.

- *Composition*: Individual statecharts usually serve as components of a more complex system. In this case, the composition of the statecharts may not define their exact execution order, causing non-determinism, too.

External non-determinism. An obvious source of non-determinism during execution is the behavior of the *environment*, i.e., the input events sent to the statechart. This is called external non-determinism, and does not cause problems like internal non-determinism. External non-determinism is observable and controllable, e.g., during simulation, it is typically handled by the user of the simulator by "impersonating" the environment, i.e., choosing the next event to send (and its parameters, if any).

2.2.4 Extended Symbolic Transition System

Extended Symbolic Transition System (XSTS) [16] is a suitable low-level formalism to describe higher-level (reactive) systems, such as statecharts. 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 thesis, I introduce XSTS based on my previous work [20].

Definition 2 (Extended Symbolic Transition System). Formally, an XSTS model is 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*, 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 individual 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 $\mathrm{var}(\varphi) = V$.

An *abstract state* of the system $s \in S = 2^C$ may cover more concrete states $s = \{c_1, \ldots, c_m\}$. We use the notation $0 \leq |s| \leq 2^{|C|}$ for the number of concrete states covered by abstract state s. Like in the case of concrete states, the notation $s(v) = x$ where $x \in D_v$ means that $c(v) = x$ in every concrete state $c \in s$ covered by s. The logical formula of an abstract state is $\varphi = ((v_1 = c_1(v_1) \wedge \cdots \wedge v_n = c_1(v_n)) \vee \cdots \vee (v_1 = c_m(v_1) \wedge \cdots \wedge v_n = c_m(v_n)))$ where $\mathrm{var}(\varphi) = V$, and it covers exactly m (maybe infinite) concrete states.

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., $IV(bool) = false$, $IV(integer) = 0$. 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 only (concrete) *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, and therefore s is a deadlock. In addition to the non-deterministic selection, transitions may be non-deterministic internally, therefore even in the case of a concrete state c, $t(c) = \{c'_1, \ldots, c'_k\}$ may yield 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 $|s'|$.

2.2.4.1 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 [16] – 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, $s' = \bigcup c'$, where $c'(v) = \varphi \wedge c'(v') = c(v')$ for every $v' \neq v \in V$ and every $c \in s$. Therefore, the number of concrete states covered by s' is less than or equal to the number of concrete states covered by s: $|s'| \leq |s|$.
- *Assumptions*: An assumption of form $[\psi]$ with ψ as a Boolean expression over the variables ($\text{var}(\psi) \subseteq V$) checks condition ψ without modifying any variable and can only be executed if ψ evaluates to *true* over the current state s, i.e., over at least one concrete state $c \in s$. In this case, the successor state s' covers the concrete states $c \in s$ covered by the original state s where ψ evaluates to *true*, therefore $|s'| \leq |s|$. Otherwise, i.e., when ψ evaluates to *false* over every concrete state $c \in s$, the set of successor states is the empty set $\emptyset$, and $|s'| = 0$.
- *Havocs*: A havoc of form $\text{havoc}(v)$ 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, $s' = \bigcup_{c \in s} \{c'_1, \ldots, c'_{|D_v|}\}$, where $c'_i(v) = D_{v_i} \wedge c'_i(v') = c_i(v')$ for every $v' \neq v \in V$ where $D_v = \{v_1, \ldots, v_{|D_v|}\}$. Therefore, $|s'| \geq |s|$.

- *Local variables*: A local variable can be declared as an operation of form *var* v_{loc} : *type* := φ.[1] 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, $V' = V \cup \{v_{\text{loc}}\}$, $s'(v_{\text{loc}}) = \varphi$, $s'(v) = s(v)$ 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:

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

- *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 default value of the type is used as an initializer unless explicitly specified by the modeler.

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.

2.2.4.2 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 my previous work [20]. In Chapter 4, I extend and formalize the splitting model transformation in order to make the executions fully explainable without changing the original semantics of the XSTS model.

2.3 Simulation

In everyday engineering work, it is useful if engineers can observe the behavior of the designed systems in an early stage of development. These observations can provide early feedback about different aspects of the system, reducing the cost of the whole development process. Due to this need, simulators are required which can simulate high-level engineering models.

With the growing complexity of the designed systems, the abstraction level of engineering models is also increasing. During the simulation of high-level models, the simulator repeatedly calculates the successor states of the system in every simulation step. At the core of the simulator, this calculation can be broken down into low-level mathematical operations. It is a challenge for the simulator to fill the abstraction gap between the actual calculations and the high-level steps.

An example of this challenge is the internal non-determinism of high-level simulation steps. An atomic high-level step may contain hidden decision points at a lower level, e.g. an atomic step of a statechart may contain a decision point if multiple transitions are fireable. For precise simulation, it is important to make these internal decision points explicit to the user, in order to give them a better understanding of the possible behaviors of the designed system.

In the case of statecharts, most simulation tools do not enable the user to simulate every possible behavior, e.g., they do not make it possible to select the exact execution order of parallel regions.

2.4 Related Tools

In my work, I extended two open source frameworks – the Gamma Statechart Composition Framework (Section 2.4.1), and the Theta Model Checking Framework (Section 2.4.2) – both developed at the Critical Systems Research Group[2] of the Department of Measurement and Information Systems, Budapest University of Technology and Economics.

2.4.1 Gamma Statechart Composition Framework

The *Gamma Statechart Composition Framework*[3] [15] is an Eclipse-based framework for the model-based design and formal analysis of component-based reactive systems. Gamma has its own modeling languages for the definition of individual *statecharts* and their *composition*. For such concepts, Gamma defines their precise formal semantics, making the use of formal methods possible.

Defining models in the Gamma Statechart Language is not the only way to use Gamma. For real-life usability, Gamma supports the integration of high-level engineering models (e.g., Yakindu and SysML statecharts) by providing X-to-Gamma model transformations. This architecture also makes the framework easily extendable with the support of further modeling languages.

Gamma statecharts support the common features of engineering-level statecharts, e.g., the modeling of *variables* with different *domains*, state refinement, and concurrent regions.

Once Gamma models are available (regardless of their origin), the framework provides numerous features.

- *Visualization*: Gamma can visualize the designed statecharts and the composite systems in graphical diagrams using PlantUML[4].
- *Validation*: Gamma provides several validation rules to statically analyze the models in design time, presenting warnings or errors to the user. These give valuable early feedback to the modeler about the possible errors of the modeled system.
- *Formal verification*: Gamma can apply formal verification on models. It provides a custom language to define the verifiable requirements (as temporal logic formulas), and transforms the models and the requirements into the input formalism of the selected formal verification backend. Currently, it supports UPPAAL [13] (extended timed automaton formalism), Theta [22] (extended symbolic transition system formalism), and SPIN [12] (Promela formalism) as formal verification backends. It also back-annotates the low-level verification result to the Gamma-level models.
- *Code generation*: Gamma can transform composite models into Java source code, keeping its component-based structure. For every Gamma-level interface, component, and system a separate Java interface or class is generated, respectively.
- *Test generation*: Gamma can generate test cases to fulfill different coverage criteria (e.g., state coverage, transition coverage). The resulting test cases are presented at the Gamma level (in the Gamma Test Language) and can also be transformed into Java code, as JUnit tests.

[2] `https://ftsrg.mit.bme.hu/en/`
[3] `https://inf.mit.bme.hu/en/gamma` and `https://github.com/ftsrg/gamma`
[4] `https://plantuml.com/`

The model transformation chains and languages of the Gamma framework are shown in Figure 2.4.

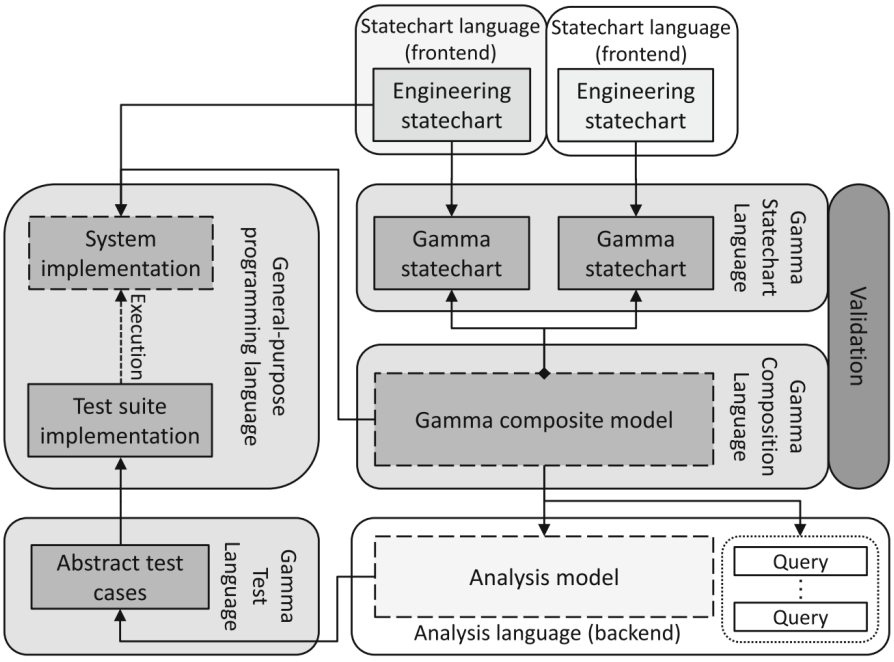

Figure 2.4: The functionalities of the Gamma framework [6].

2.4.2 Theta Model Checking Framework

The Theta Model Checking Framework[5] [22] is a generic, modular, and configurable model checking framework. The architecture of Theta is shown in Figure 2.5.

Theta can be divided into four main layers.

- *Formalisms*: Theta can handle multiple formalisms as input, such as *timed automata*, *control flow automata*, and *transition systems*. Due to its modular architecture, further formalisms can be integrated. Every input model is transformed into a common inner abstract state space representation, regardless of the original input formalism. In this work, I focus on the *Extended Symbolic Transition System* (XSTS) [16] formalism as the input of Theta.

- *Analysis back-end*: The analysis back-end provides the verification algorithms themselves. It contains formalism-specific *interpreters* providing a common interface towards the algorithms by defining three functions: the *init function* calculates the initial abstract states of the system, the *transfer function* calculates the successor states of a given state, and the *action function* calculates the available actions in a given abstract state. The verification algorithms usually use abstraction for which various *abstract domains* are defined.

- *SMT solver interface*: Many components rely on *satisfiability modulo theories* [2] (SMT) solvers for which a general interface is provided by Theta. It enables the use

[5] `https://inf.mit.bme.hu/en/theta` and `https://github.com/ftsrg/theta`

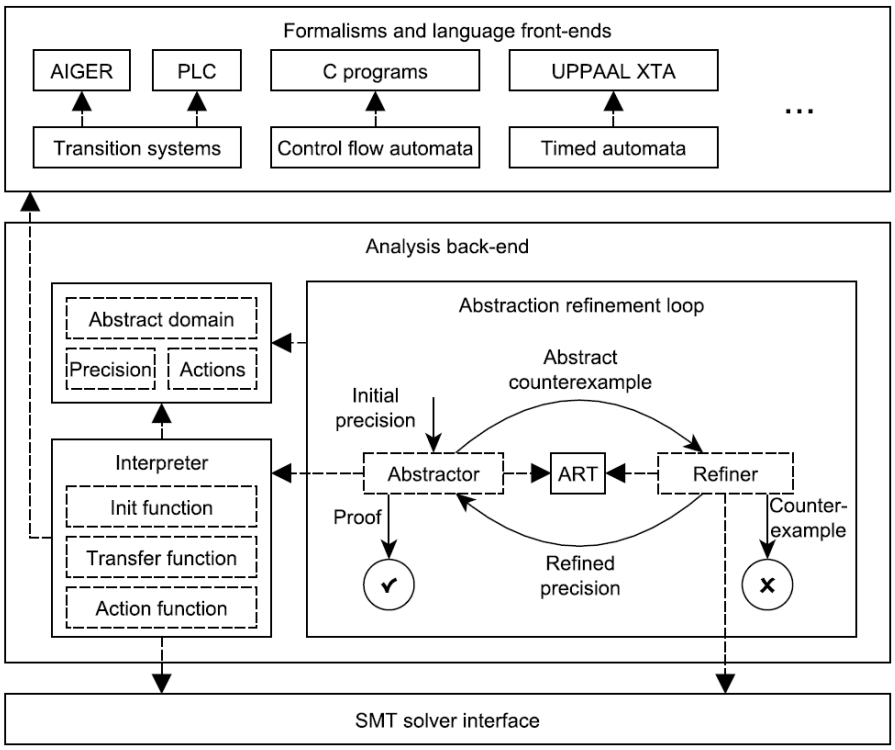

Figure 2.5: The architecture of the Theta framework [22].

of different SMT solvers, and currently, it fully supports the Z3[6] solver developed by Microsoft.

- *Tools*: Theta provides simple command-line applications which are responsible for reading the input and then instantiating and calling the algorithm specified by the arguments.

The analysis is done by the *counterexample-guided abstraction refinement* (CEGAR) loop in the analysis back-end, Its central data structure is the *abstract reachability graph* [10] (ARG) or *abstract reachability tree* (ART). Its nodes represent abstract states, and its edges are labeled with actions. The ARG is modified by the two main parts of the loop, the *abstractor* and the *refiner*.

The initial ARG is constructed by the abstractor, based on the initial precision which comes from the selected algorithm and the formalism-specific interpreter. If there is no counterexample in the ARG, it *proves* the correctness of the model. Otherwise, if an abstract counterexample exists, the refiner checks its feasibility. If the abstract counterexample is feasible, i.e., real, the model is incorrect, and the counterexample is returned. Otherwise, the refiner refines the abstraction of the ARG to eliminate that spurious counterexample.

[6] `https://github.com/Z3Prover/z3`

Chapter 3

Requirements on the Simulation Framework

In this chapter, I declare the main requirements on the simulation framework. The requirements cover semantics (Section 3.1), variable values (Section 3.2), back-annotation (Section 3.3), non-determinism (Section 3.4), control (Section 3.5), and observation (Section 3.6). I analyze the requirements, and as a result, present some design decisions and the high-level architecture of the simulation framework (Section 3.7).

The requirement diagram of the simulator is shown in Figure 3.1.

3.1 Precise Formal Semantics

REQ1. *The simulator must be able to simulate component-based reactive systems, following the precise formal semantics of the models.*

Component-based reactive systems can be modeled with Gamma statecharts. Although Gamma statecharts have precise formal semantics, their direct simulation would be hard due to their complexity and high-abstraction level. For formal verification purposes, Gamma provides a semantics-preserving Gamma-to-XSTS model transformation which can be reused for simulation, too. The simulation of XSTS models is easier, due to their lower abstraction level.

In order to follow the exact semantics of XSTS models during simulation, I decided to reuse the model checker infrastructure of Theta, so exactly the same code calculates e.g., the successor states of a state during the simulation and formal verification. With this approach, I can save a lot of unnecessary work and avoid redundant code bases, resulting in easier maintainable software components.

3.2 Variable Values

REQ2. *The simulator must be able to present the exact value of every variable of the system at every step.*

An important use case of the simulation of behavioral models is that engineers can try out the models they designed. The more details a simulator can give to its user, the more useful it is. It is crucial to always be able to present the simulation state to the user,

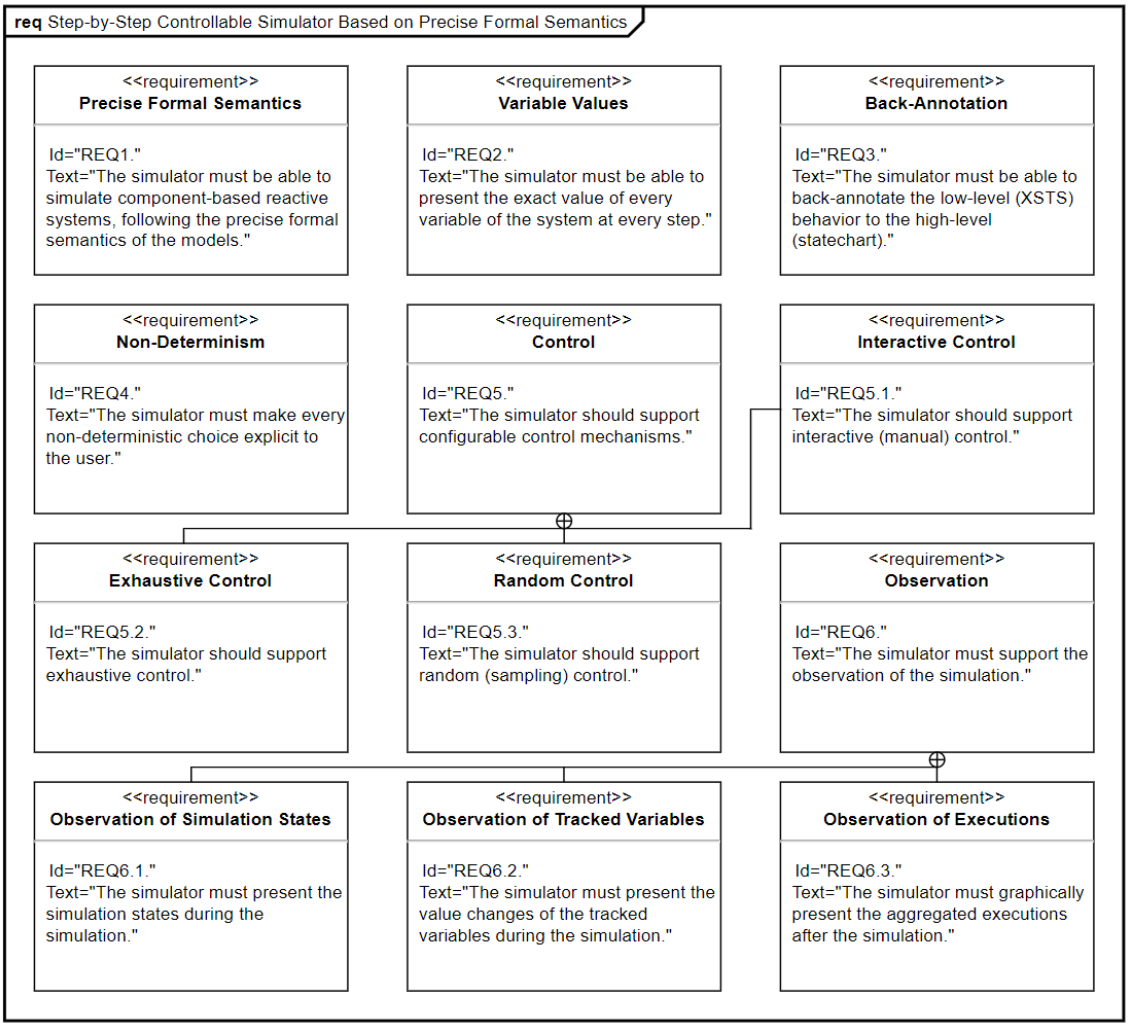

Figure 3.1: Requirement Diagram of the simulator.

which, in the case of XSTS models, consists of the exact value of every variable of the system including variables encoding the state of execution (control variables).

The model checking infrastructure of Theta supports the usage of abstraction-based algorithms, such as counterexample-guided abstraction refinement (CEGAR). Abstraction improves the efficiency of model checking, which leads to the support of more complex models. In the case of simulation, efficiency is not a key requirement, so instead of any kind of abstraction, the simulator makes the underlying model checker infrastructure explicitly track the value of every variable.

3.3 Back-Annotation

REQ3. *The simulator must be able to back-annotate the low-level (XSTS) behavior to the high-level (statechart).*

Although the simulator simulates the statecharts by generating a semantics-preserving XSTS model from them and simulating the XSTS model itself, it is expected to observe the behavior of the original statechart model, not the XSTS model. To fill the abstraction gap, proper back-annotation could be provided to map every change in the low-level model state to a higher-level behavior, but this is beyond the scope of this work.

Instead, the back-annotation in my work relies on the log mechanism of Gamma. In the high-level statecharts, explicit *log* statements can be defined with a custom string literal. During the Gamma-to-XSTS model transformation, the log statements of every component are transformed into lower-level model elements.

- For every *component type*, an XSTS enum type is created, with the same literals as the literals used in the log statements of the component, and a default literal *None*.
- For every *component instance*, an XSTS variable is created, with the enum domain corresponding to its component type, and the default value *None*.
- For every *log statement*, an XSTS assignment is created, which assigns the enum literal corresponding to the log literal, to the variable corresponding to the component instance. Then, the variable needs to be reset so the default value *None* is assigned to it.

During the simulation of the low-level XSTS model, the value changes of these log variables can be observed to follow the execution of the high-level statechart models.

Example 3 (Transformation of log statements). *Consider a component type T, and component instances a and b of type T. In T (e.g., in the high-level statechart transitions), log statements are defined with string literals log1 and log2.*

During the Gamma-to-XSTS model transformation, the following log-related model elements are created:

- *For component type T, an XSTS enum type $D_{\log_T}$ is created with literals None, log1, and log2.*
- *For component instances a and b, XSTS variables $\log_a$ and $\log_b$ are created, with the same domain $D_{\log_T}$. The initial value of every log variable is the default log literal None.*

- *E.g., in the case of component instance a, the log statement with string literal log1 is transformed to XSTS assignments* $log_a := log1$*, and* $log_a := None$*.*

3.4 Non-Determinism

REQ4. *The simulator must make every non-deterministic choice explicit to the user.*

In behavioral models, there may be decision points, where the execution can continue in several directions. These decisions are modeled as non-deterministic decisions, which is especially common in the case of reactive systems, where the events coming from the environment are usually modeled as non-determinism.

During simulation, our goal is to provide full control for the user, to be able to control every step of the simulation. Therefore, every non-deterministic decision point must be made explicit to the controller of the simulation.

In XSTS models, there can be non-deterministic decisions inside atomic transitions, too. In order to provide full control for the user, even these inner non-deterministic decisions should be recognized and made explicit. In the given model checker environment, it is not possible to control the inner steps inside an atomic transition, so a pre-process model transformation step (called *splitting*) is required to eliminate internal non-determinism. The splitting algorithm is detailed in Chapter 4.

3.5 Control

REQ5. *The simulator should support configurable control mechanisms.*

For different use cases, different control strategies are suitable. At a non-deterministic decision point (i.e., when there are several successor states), the controller of the simulator should select a successor state. Depending on the goal of the simulation and the controller of the simulator (which can be a person interacting with the simulator, or some automatic code), different control mechanisms are required.

3.5.1 Interactive Control

REQ5.1. *The simulator should support interactive (manual) control.*

When the goal is to let the modeler *try out* their model, they would obviously like to control the simulation themselves. This means, that the simulator is interactive: the decision points are made explicit to the user, who can manually select from the possible successor states, i.e., they can manually select the simulated behavior from the possible ones.

3.5.2 Exhaustive Control

REQ5.2. *The simulator should support exhaustive control.*

When the goal is to traverse *every possible execution* of a model, interactive control is not practical. Instead, an automated approach is necessary, which – following some search strategy systematically – can guarantee the completeness of the traversal.

Note, that the possible executions of a model can be represented as a graph, where the nodes are the states of the execution, and the edges are the possible paths of the execution. In order to explore every possible execution of a model, graph traversal algorithms are suitable, e.g., depth-first search (DFS). This approach is similar to model checking but not for checking a property.

3.5.3 Random Control

REQ5.3. *The simulator should support random (sampling) control.*

When the goal is to traverse the possible executions of a model, exhaustive traversal is not always an option due to its complexity. Note, that the number of possible executions depends exponentially on the number of non-deterministic decisions in the model. Therefore, in the case of complex models with many non-deterministic constructs (e.g., orthogonal regions), another approach is required to explore the possible executions of a model, not in a complete but in a systematic way.

To achieve this, the simulator can select randomly from the possible successor states at every decision point. In this case, we can increase the completeness of our traversal by increasing the number of simulations.

3.6 Observation

REQ6. *The simulator must support the observation of the simulation.*

It is crucial for the user to be able to observe the simulation – basically, this is the goal of the simulation. This observation is required at two levels:

- *During the simulation*, the simulation states are presented to the user.
- *After the simulation*, the whole execution(s) is/are presented to the user.

3.6.1 Observation of Simulation States

REQ6.1. *The simulator must present the simulation states during the simulation.*

During the simulation, the simulator presents every simulation state to the user. A simulation state is represented by the exact value of every (control) variable of the XSTS model. In the case of interactive simulation, the user can understand the execution and control the decisions based on the presented simulation states. In the case of automated simulation, the presented simulation states can be analyzed and processed programmatically.

3.6.2 Observation of Tracked Variables

REQ6.2. *The simulator must present the value changes of the tracked variables during the simulation.*

An XSTS model usually contains many variables, which makes the analysis of whole simulation states complex. It is a common use case to track the value of only a few variables, especially the changes in their values. Therefore, the user can track a subset of the XSTS variables, and the simulator explicitly presents their value changes. Observing

only the value changes of some tracked variables helps the user to focus only on a specific aspect of the model execution.

Note, that this is especially useful in the case of log variables. If the user tracks the low-level log variables, the presented value changes correspond to the executions of the high-level log statements. This approach allows the user to control the granularity of the observation, by inserting/removing high-level log statements and selecting the tracked low-level log variables.

3.6.3 Observation of Executions

REQ6.3. *The simulator must graphically present the aggregated executions after the simulation.*

During the simulation, the simulator collects the value changes of tracked variables, and the executions are represented by the sequence of these value changes. After the simulation finishes, the simulator aggregates the executions (if there are more) and presents them graphically.

In the case of a single execution (in interactive mode), no aggregation is necessary. In other cases, the same parts of different executions are merged, in order to emphasize the *differences* between the different executions.

Note, that the granularity of the executions' graphical representation can be controlled by selecting the set of tracked variables. The more variables the user track, the more value changes will be presented and the more insights the user will get about the executions.

3.7 Architecture

From the above-described requirements, I designed the architecture of the simulator framework which is shown in Figure 3.2, an informal figure, showing the main components and interactions of the simulator. It also shows which components satisfy the above-defined requirements (requirements are marked with gray boxes, satisfy relations marked with dashed lines).

The XSTS splitter splits the original XSTS model into a split one, which the simulator simulates – it is essential that the model is split because it makes every non-determinism observable and controllable. The core of the simulator is the existing model checker infrastructure of Theta, which is responsible for calculating the possible successor states of the current simulation state. This is wrapped by the simulator, which handles the interactions with the user through a well-defined interface.

The user can control the simulation (at decision points), and the simulator can provide information about different aspects of the simulation to the user. While these pieces of information flow through an interface, different implementations can be used for different control (see Section 3.5) and observation (see Section 3.6) purposes.

The splitting algorithm of XSTS models and its implementation are detailed in Chapter 4. The architecture and implementation of the simulator are detailed in Chapter 5.

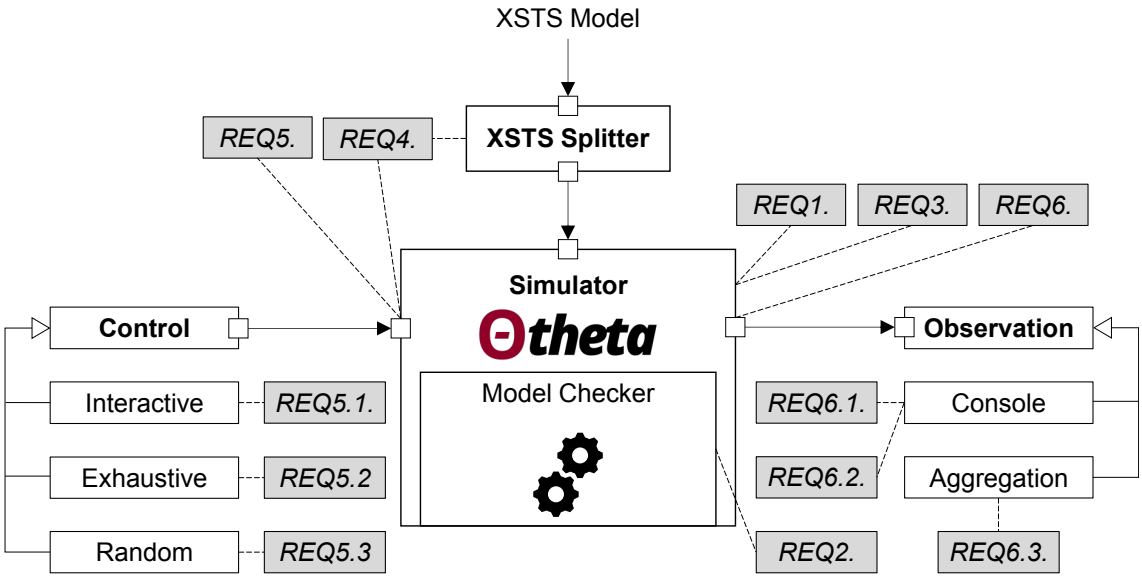

Figure 3.2: Architecture of the simulator.

Chapter 4

Splitting XSTS Transitions

In this chapter, I present the splitting of XSTS transitions, a model transformation to eliminate internal non-determinism from XSTS models. I declare the necessity of splitting (Section 4.1) and give an overview of the different kinds of non-determinism in XSTS models (Section 4.2). I formally detail the exact rules of splitting the different XSTS operations (Section 4.3) and present an approach to merge all the transition relations of an XSTS model into a single one (Section 4.4). Finally, I detail the implementation of the splitting algorithm (Section 4.5).

4.1 Motivation

During the execution of an XSTS model, several non-deterministic decisions can occur. Since the goal of this work is to provide a step-by-step controllable simulator, every non-deterministic choice has to be made explicit to the controller of the simulation to satisfy **REQ4.**

To achieve this goal, first of all, the simulator itself has to observe these non-deterministic decision points. Due to the atomicity of XSTS transitions (and the underlying model checker infrastructure), this observation can not be done inside them, so the original internal non-determinism inside transitions has to be eliminated.

This can be reached by a pre-processing model-transformation step, which breaks down the non-deterministic transitions into smaller, deterministic parts (*micro-steps*), without changing the original behavior of the model – so the original *control flow* [1] of the micro-steps does not change. Preserving the original semantics is essential in order to satisfy **REQ1.**

I originally introduced the splitting of XSTS transitions in [20].

4.2 Non-Determinism in XSTS Models

XSTS models can have different kinds of non-deterministic decision points.

- *External non-determinism*: The execution of a transition set T means the execution of a randomly selected transition $t \in T$ from it. This transition selection is non-deterministic, but this is outside the atomic transitions, so it occurs in a stable state of the system.

- *Internal non-determinism*: The execution of some operations is non-deterministic (e.g., choice, parallel). If a transition t contains a non-deterministic operation inside, due to the atomicity of t, we can not fully explain the non-deterministic decision made during the execution of t, based on the next successor stable state.

Example 4 shows the problem of controllability, and Example 5 shows the problem of observability, both caused by internal non-determinism.

Example 4 (Internal non-determinism with different target states). *Consider the following transition:*

$$t = \begin{pmatrix} x := 0, \\ (x := x + 1 \ \mathit{or} \ x := x + 2) \end{pmatrix}$$

The execution of t can result in two stable states, where $x = 1$, and where $x = 2$. Although we can observe the different states, due to the atomicity of t, after selecting it to fire, we can not control its internal behavior.

Example 5 (Internal non-determinism with a single target state). *Consider the following transition:*

$$t = \begin{pmatrix} x := 0, \\ (x := x + 1 \ \mathit{or} \ x := x + 2), \\ (([x = 1], x := x + 2) \ \mathit{or} \ ([x = 2], x := x + 1)) \end{pmatrix}$$

While every execution of t will result in the only stable state, where $x = 3$, after the execution, we can not explain what happened exactly inside *the execution of the transition: which happened first, the $x := x + 1$ or the $x := x + 2$.*

In order to be able to fully control and observe non-deterministic decisions, internal non-determinism should be made external.

4.2.1 Internal Non-Determinism

Inside an XSTS transition, 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.
- *Parallels*: In every step of a parallel action $op_1 \ || \ \ldots \ || \ op_n$, 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. Havocs are for modeling the non-deterministic inputs from the environment, and this non-determinism can not be made external. However, it is possible to split every havoc into a separate transition, so that the selected value is exactly observable in the successor state of the transition.

Although *conditionals* do not cause internal non-determinism, since the evaluation of the condition ψ of a conditional $(\psi) \ ? \ op_{\text{then}} \ : \ op_{\text{else}}$ is deterministic, the splitting of ψ, op_{then} and op_{else} makes the observation of the actual execution easier.

In general, splitting is a *model-transformation* defined at the following levels:

- The splitting of an *XSTS model* $\langle V, Tr, In, En \rangle$ splits every transition set Tr, In, En, resulting in a split XSTS model $\langle V', Tr', In', En' \rangle$.
- The splitting of a *transition set* T means the splitting of every transition $t \in T$.
- The splitting of a *transition* t breaks it down into smaller transitions (*fragments)*, yielding at least one split transition, so the split transition set T' will contain at least as many transitions as the original T: $|T'| \geq |T|$.

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: $|s'| \leq 1$ – except the transitions containing a havoc statement, in which case it is impossible.

4.3 Splitting Rules

In the following, we formalize the splitting of a transition relation T by defining splitting rules of *sequences*, *havocs*, *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): $\text{split}(t) = \{t'_1, \ldots, t'_n\}$. The splitting of a transition relation T results in the union of the fragments of every original transition $t \in T$: $\text{split}(T) = \bigcup_{t \in T} \text{split}(t)$.

Definition 3 (Fragment). 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, $\text{frag}(op, x, y) = ([pc = x], op, pc := y)$, 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. A state is called *stable*, if $pc = 0$, otherwise it is called a *pseudostate*.

In the following, we use $\text{split}(op, x, y)$, 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 $\text{split}(t) = \text{split}(t, 0, 0)$.

The *splittable* operations are havocs, choices, conditionals, and parallels.

4.3.1 Sequence

In case of a sequence $seq = op_1, \ldots, op_n$, if op_i is the first splittable operation, the resulting fragments will be $\text{split}(seq, x, y) =$

$\{(\text{frag}((op_1, \ldots, op_{i-1}), x, \xi_1), \text{split}(op_i, \xi_1, \xi_2), \text{split}((op_{i+1}, \ldots, op_n), \xi_2, y)\}$, where ξ_i denotes a unique program counter value, which can be generated incrementally, for example. If there is no splittable operation in seq, $\text{split}(seq) = \text{frag}(seq, x, y)$.

Example 6 shows a sequence without any splittable operation, so no actual splitting is performed. I show more complex examples in the following.

Example 6 (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:*

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

In order to observe every value change of the tracked variables, it is necessary to split every assignment to them into separate fragments. It can be simply achieved by treating these assignments as splittable operations, like havocs (see Section 4.3.2).

4.3.2 Havoc

The splitting of a *havoc* of form $h = \text{havoc}(v)$ wraps this single basic operation into a separate fragment. Formally, $\text{split}(h, x, y) = \text{frag}(h, x, y) = \{([pc = x], \text{havoc}(v), PC := y)\}$.

Although the internal non-determinism caused by havoc can not be made external, moving it to a separate split transition is beneficial. This separate transition *only* models a non-deterministic assignment, which can be made explicit to the user, letting them choose the actual value.

Example 7 (Splitting sequence with havoc). *For transition $t = (x := 1, \text{havoc}(y), z := y)$ with a splittable havoc, splitting will result in 3 fragments:*

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

4.3.3 Choice

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, $\text{split}(ch, x, y) = \bigcup_{i=0}^{n} \text{split}(op_i, x, y)$.

Example 8 (Splitting choice). *For transition $t = (x := 1 \text{ or } x := 2)$ with a splittable choice with 2 branches, splitting will result in 2 fragments:*

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

Example 9 (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:*

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

4.3.4 Conditional

The splitting of a *conditional* of form $cond = (\psi)\ ?\ op_{\text{then}}\ :\ op_{\text{else}}$ 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, $\text{split}(cond, x, y) = \{\text{condfrag}(\psi, x, \xi_{\text{then}}, \xi_{\text{else}})\} \cup \text{split}(op_{\text{then}}, \xi_{\text{then}}, y) \cup \text{split}(op_{\text{else}}, \xi_{\text{else}}, y)$.

The condition fragment $\text{condfrag}(\psi, x, \xi_{\text{then}}, \xi_{\text{else}})$ checks $pc = x$, then assigns ξ_{then} or ξ_{else} to *pc* based on the evaluation of ψ, repsectively. Formally, $\text{condfrag}(\psi, x, \xi_{\text{then}}, \xi_{\text{else}}) = ([pc = x], pc := (\psi)\ ?\ \xi_{\text{then}}\ :\ \xi_{\text{else}})$, 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 10 (Splitting conditional). *For transition* $t = ((x > 0)\ ?\ y := x\ :\ y := 0)$ *with a splittable conditional, splitting will result in 3 fragments:*

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

4.3.5 Parallel

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 $fork$*ing* and $join$*ing* 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, $\text{split}(par, x, y) = \{\text{forkfrag}(x, \xi, \bigcup_{i=1}^{n} pc_i), \bigcup_{i=1}^{n} \bigcup_{j=1}^{|op_i|} \text{parfrag}(\xi, pc_i, op_i, j), \text{joinfrag}(\xi, y, \bigcup_{i=1}^{n} pc_i)\}$.

The $fork$ fragment $\text{forkfrag}(x, \xi, PC)$ 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, $\text{forkfrag}(x, \xi, PC) = ([pc = x], seq_{i=1}^{|PC|} PC_i := 1, pc := \xi)$, 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 $\text{parfrag}(\xi, pc_i, op_i, j) = \bigcup f'$. 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.

I 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. This difference is important for preserving the original semantics of *parallels* – using the original split, would not enable every interleaving of the branch executions, by making the execution of non-splittable sequences atomic.

The *join* fragment joinfrag(ξ, 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 11 (Splitting parallel). *For transition* $t = ((x := 1, y := x) \;||\; (x := 2, y := x))$ *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:*

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

4.3.6 Local Variables

Splitting can cause the declaration and usage of a local variable to end up in different transitions (fragments). Therefore, some local variables may become global and management of the scope has to be emulated with additional operations.

After the splitting of a transition relation T, a post-process step is needed to fix the broken local variables whose declaration and usage ended up in different fragments.

Definition 4 (Usage relation of local variables). In case of a transition relation T, $U \subseteq V_\ell \times T'$ is the *usage relation* of the local variables, where V_ℓ is the set of local variables declared in any $t \in T'$, and T' is the transition relation after splitting ($split(T) = T'$). ▪

We have $(v_\ell, t) \in U$ *iff* transition t uses local variable v_ℓ either in

- a local variable declaration $var\ v_\ell : D_{v_\ell}$,
- an assignment $v_\ell := \varphi$,
- an expression φ (either in an assumption or the right-hand-side of an assignment or declaration), or
- a havoc havoc(v_ℓ).

Definition 5 (End fragments of local variable scope). As defined in Section 2.2.4.1, the *scope* scope(v_ℓ) of a local variable v_ℓ is its direct container composite operation. During splitting, this scope may be split into several fragments F: split$(\text{scope}(v_\ell), x, y) = F = \{f_1, \ldots, f_n\}$. The *end fragments* F_E of scope sc are defined

as the subset of fragments $F_E \subseteq F$ which end with an assumption $[pc = y]$. We use the notation $F_E(v_\ell)$ for the set of end fragments of the scope of local variable v_ℓ.

Due to our previous assumptions, the scope of v_ℓ is always a sequential action ending with a composite or a basic operation op_n. If op_n is a basic operation, it is the only end of the scope, so $|F_E(v_\ell)| = 1$. If op_n is a composite operation with more branches (e.g., choice, parallel), the ends of the scope consist of the end(s) of every branch (which may have more branches, recursively). ▪

For every local variable $v_\ell \in V_\ell$ used by more fragments ($(v_\ell, t_1) \in U$ and $(v_\ell, t_2) \in U$ and $t_1 \neq t_2$), we execute the following steps:

1. Make the local variable *global* by removing the local variable declaration and adding v_ℓ to V'. This means, that v_ℓ will be in V' permanently, not only during its original scope. The original initial value $IV(v_\ell) = \varphi$ of v_ℓ is replaced with the initial value of its type D_{v_ℓ}, so as a global variable, it will be initialized to $IV(D_{v_\ell})$.

2. Add an assignment $v_\ell := \varphi$ to the original place of the local variable declaration (if $\varphi \neq IV(D_{v_\ell})$, i.e., the original initial value is not the initial value of its type).

3. Append an assignment $v_\ell := IV(D_{v_\ell})$ to every end fragment $f_E \in F_E(v_\ell)$ of the scope of v_ℓ to reset the variable. This transformation guarantees that $v_\ell = IV(D_{v_\ell})$ outside of the original scope of v_ℓ. This is not required by the semantics but helps in reducing the state space.

Example 12 (Splitting local variable usages). *For transition* $t = (v_\ell : integer := 1, ((v_\ell := v_\ell + 1)$ *or* $(v_\ell := v_\ell + 2)), [v_\ell > 1])$ *with a local variable* v_ℓ *used by both of the branches of a choice, splitting will result in 4 fragments, while* v_ℓ *will be made global, i.e., added to* V' *with initial value* $IV(v_\ell) = IV(D_{v_\ell}) = IV(integer) = 0$*:*

$$\text{split}(t) = \text{split}(t, 0, 0) = \begin{Bmatrix} ([pc = 0], & v_\ell := 1, & pc := 1), \\ ([pc = 1], & v_\ell := v_\ell + 1, & pc := 2), \\ ([pc = 1], & v_\ell := v_\ell + 2, & pc := 2), \\ ([pc = 2], & [v_\ell > 1], v_\ell := 0, & pc := 0) \end{Bmatrix}$$

Note, that the original initial value 1 *of* v_ℓ *is set at the place of the original local variable declaration, and at the end of its original scope,* v_ℓ *is set back to* 0*, which is the initial value of its type.*

4.4 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 $t \in \text{split}(In) \cup \text{split}(En)$ into $\text{split}(Tr)$, 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*: $V' = V \cup \{init, trans\}$, $D_{init} = D_{trans} = bool$, $IV(init) = true$, $IV(trans) = false$.

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, certain assumptions and assignments are necessary. The assumptions are needed at the beginning of every split fragment, while the assignments are only needed at the end of the original non-split transitions. To achieve this, the following steps are required:

1. We extend every original non-split transition $t = op, t \in In \cup En \cup Tr, op \in Ops$ with the following assignments, resulting in t':
   - $t \rightsquigarrow t' = (op, init := false)$ for every $t \in In$
   - $t \rightsquigarrow t' = (op, trans := true)$ for every $t \in En$
   - $t \rightsquigarrow t' = (op, trans := false)$ for every $t \in Tr$
2. We split every transition relation (containing the t' transitions extended in the previous step), resulting in split transition relations split(In), split(En), and split(Tr)
3. We extend every split fragment $f = op, f \in \text{split}(In) \cup \text{split}(En) \cup \text{split}(Tr), op \in Ops$ with the following assumptions, resulting in f':
   - $f \rightsquigarrow f' = ([init], op)$ for every $f \in \text{split}(In)$
   - $f \rightsquigarrow f' = ([\neg init \wedge \neg trans], op)$ for every $f \in \text{split}(En)$
   - $f \rightsquigarrow f' = ([\neg init \wedge trans], op)$ for every $f \in \text{split}(Tr)$

After these transformations, we can merge the f' 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 13 (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 $En = \{en_1, en_2\}$ (all transitions are non-splittable) the merged XSTS model will be $\langle V', Tr', In', En' \rangle$, where $V' = V \cup \{init, trans\}$, $In' = En' = \emptyset$, and*

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

4.5 Implementation

I implemented splitting as an extension of the Gamma Statechart Composition Framework. Gamma tasks (such as a Gamma-to-XSTS transformation, called an *analysis* task) can be defined in *.ggen* files, based on an *Xtext*[1] grammar and an *Eclipse Modeling Framework* (EMF)[2] metamodel. I added a configuration option to the Gamma-to-XSTS transformation for splitting.

Currently, the implementation supports two splitting configurations:

- *NONE* (default): no transition is split at all.
- *CHOICE*: every transition is split by choices, conditionals, parallels, and havocs (see Section 4.3) and every transition relation is merged into Tr (see Section 4.4).

Although these two options could be modeled with a simple boolean, for future extendability, I added an enum type AnalysisSplit to the EMF metamodel with literals NONE and CHOICE. I extended the AnalysisModelTransformation class (which represents an analysis task defined in a .ggen file) with a field split type of AnalysisSplit, and extended the grammar rules, respectively. The textual syntax of an example analysis model transformation task with a split option is shown in Listing 4.1.

```
analysis {
    component: System
    language: Theta
    split-by: CHOICE
}
```

Listing 4.1: Example analysis model transformation task with a split option in a .ggen file.

I extended the existing execute method of the GammaToXstsTransformer class with a conditional call (based on the value of the split field) to the split method of the newly created XstsSplitter class in which I implemented the splitting algorithm. Basically, splitting is a post-process step after the existing Gamma-to-XSTS model transformation.

The XstsSplitter class is written in *Xtend*[3] which is a modern dialect of Java introducing some language elements to make coding more efficient, such as *dispatch* methods. According to the Xtend documentation:[4]

"For a set of visible dispatch methods in the current type hierarchy with the same name and the same number of arguments, the compiler infers a synthetic dispatcher method. This dispatcher uses the common supertype of all declared arguments. The method name of the actual dispatch cases is prepended with an underscore and the visibility of these methods is reduced to protected if they have been defined as public methods. Client code always binds to the synthesized dispatcher method."

Dispatch methods provide a convenient way to implement some type-based logic for different types of a type hierarchy. In the case of splitting, the splitting rules are implemented as *dispatch* methods for every XSTS operation type. This approach is easily extendable and configurable with further options.

[1] https://www.eclipse.org/Xtext/

[2] https://www.eclipse.org/modeling/emf/

[3] urlhttps://www.eclipse.org/xtend/

[4] https://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html#polymorphic-dispatch

I also added some new annotations to the XSTS metamodel in order to denote whether an XSTS model is split or not, and whether its transition relations are merged or not.

- SplitAnnotation denotes that the XSTS model is split, i.e., transitions do not contain internal non-determinism.
- NoEnvAnnotation denotes that En and In are empty, i.e., only Tr contains non-empty transitions.

The textual syntax of the annotations is shown in Listing 4.2.

```
//@split
//@noenv
...
trans {
    ...
}
init{
}
env{
}
```

Listing 4.2: Example split XSTS model with annotations.

Chapter 5

Simulation Framework

In this chapter, I detail the simulation framework itself for the step-by-step controllable simulation of split XSTS models. I describe the basics of the simulation algorithm and present how it can be used to simulate a single execution of a model (Section 5.1). Then, I extend the previous solution to be able to simulate several executions of a single model, e.g., for exhaustive simulation (Section 5.2). I describe the interactions between the simulator and the user, focusing on the different control and observation opportunities (Section 5.3). Finally, I detail the implementation of the simulation framework (Section 5.6).

Note that Section 5.1 focuses only on the simulation itself, i.e., how to reuse a model checker to calculate the successor states in a simulation framework. Then, Section 5.2 extends the previous algorithm for exhaustive simulation and formalizes the representation of executions as execution traces. These sections do not dive into the interaction aspects (control, observation) of the simulator, instead, they are separately detailed in Section 5.3.

5.1 Single Simulation

From a high-level perspective, the simulator starts from the initial state of the given split XSTS model. At any state, the simulator calculates the possible successor states by using the underlying model checking infrastructure. If there are more successor states, the simulator asks the user to select from the possible successor states.

Then, the simulation state is changed to the selected successor state. This simulation loop continues until there is no successor state (i.e., the execution of the model is finished) or the selected successor state is covered (i.e., the simulation already visited that state). This basic algorithm is shown in Algorithm 1.

The usage of an existing model checking framework to calculate the possible successor states guarantees to follow the precise formal semantics of the models so satisfies **REQ1.** During the calculation of the successor states, the simulator (i.e., the underlying model checking infrastructure) does not use any abstraction over the variables to satisfy **REQ2.**

Theta transforms every input formalism into a common internal representation, *Abstract Reachability Graph* (ARG) [10]. 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. I introduce *Reachability Graph* to represent the state space during the algorithm.

Definition 6 (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 another node $n' \in N$.

Algorithm 1: SingleSimulate Simulating a single execution of a split XSTS model in high-level.

```
   Input: Split XSTS model XSTS = ⟨V, Tr, In, En⟩ with initial state c_0
 1 SingleSimulate (XSTS)
 2     N ← {n(c_0)}, E ← ∅, C ← ∅
 3     RG ← ⟨N, E, C⟩
 4     successors ← N
 5     while |successors| > 0 do
 6         node ← successor ∈ successors
 7         Close(node, RG)
 8         successors ← ∅
 9         if node is not covered then
10             successors ← Expand(node, RG)
11     return
```

In Algorithm 1, RG is a *reachability graph*, which is built by the Close and Expand methods – these methods are existing parts of the model checking infrastructure. 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.

In order to satisfy **REQ5.1.** (Interactive Control), the selection of the next successor state (see Line 6 of Algorithm 1) can be made explicit to the user. Thus, the user can interactively control the simulation at every decision point. **REQ5.3.** (Random Control) can be simply satisfied by replacing the interactive successor selection with a random selection.

5.2 Exhaustive Simulation

In order to satisfy **REQ5.2.** (Exhaustive Control), it is necessary to be able to simulate more executions of a model. For this purpose, the single simulation (presented in Algorithm 1) is wrapped by another loop, which controls the reiteration of the simulation.

After the simulation reached its end, the user can tell the simulator to backtrack to the last decision point, where at least one uncovered successor state exists. Then, the simulator restores that previous simulation state and continues the simulation in another direction. The traversal algorithm of every execution is shown in Algorithm 2.

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 7 (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 8 (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.

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'.

$\sigma_{P_{\text{last}}}$ denotes the last element (node) of σ_P. $\Sigma_P \leftarrow \Sigma_P \cup \{\text{copy}(\sigma_P)\}$ 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 2: ExhaustiveSimulate Simulating 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  ExhaustiveSimulate (XSTS)
2      Σ_P ← ∅, σ_P ← [ ]
3      N ← {n(c_0)}, E ← ∅, C ← ∅
4      RG ← ⟨N, E, C⟩
5      successors ← N
6      traverse ← true
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         Σ ← Σ ∪ {copy(σ)}
16         if ∃n_{i_max} ∈ σ_P with unexpanded successors then
17             successors ← UnexpandedSuccessors(n_{i_max}, RG)
18         else
19             traverse ← false
20     Σ ← Σ_P as executions
21     return Σ
```

5.2.1 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 9 (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, $ET \leftarrow [ET, \bigcup_{\forall v_T \in V_T:\ \text{last}(v_T) \neq c(v_T)}(v_T, c(v_T))]$ where $\text{last}(v_T)$ 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. With the usage of high-level log statements and the tracking of the corresponding low-level variables, the high-level behaviors can be observed during the simulation, so the simulator satisfies **REQ3.**

5.3 Interactions

During the simulation, the simulator communicates with the user (modeled as a SimulatorListener) through an interface. For different communication purposes, a specific SimulationState object is constructed from different aspects of the state of the simulation and passed to the listener. Every SimulationState object contains the exact value of every variable of the system. (**REQ2.**, **REQ6.1.**) In some cases, when the communication is two-way (i.e., an answer is required from the user), the SimulationState objects are mutable, so the user can write the answer into them.

These interactions cover the following:

- simulationStarted: The simulator notifies the listener that the simulation has started. No answer is required.
- oneSuccessor: If there is only one possible successor state, the simulator sends it to the listener. The simulator sends the current simulation state (i.e., the values of the variables) and some trace information about the only fireable transition (e.g., the line and column numbers of the transition in the textual XSTS file). No answer is required.
- moreSuccessors: If there are more possible successor states, i.e., there are more fireable transitions, the simulator sends them (in the same format, like oneSuccessor) to the listener. The listener must answer this, by selecting a transition to fire – identified by its index. (**REQ4.**)
- askForConcreteValue: If a transition with a havoc statement leads to the selected successor state, the simulator asks the listener for the concrete value to be assigned to the variable – so instead of a non-deterministic assignment by the simulator, it is the listener's responsibility to select a value from the corresponding domain.
- successorSelected: If a transition, which leads to the selected successor state, does not contain a havoc statement and there were more possible successor states to select from, the simulator notifies the listener about the actually selected successor state. No answer is required.
- valueChanged: At every state, the simulator checked for every tracked variable whether their value has changed. If the value of any tracked variable has changed,

the simulator notifies the listener about their new value. (**REQ6.2.**) No answer is required.

- splitTransitionCommitted: The simulated XSTS model is split, so it has *pseudostates*, which are not real states of the original system – they just occurred because of the split of the originally atomic transitions. Thus, the simulator notifies the listener, when the current simulation state is a *real* state of the original XSTS model, i.e., not a pseudostate. No answer is required.
- simulationEnded: The simulator notifies the listener when the simulation of the current execution has ended. The listener can make the simulator backtrack to the last decision point with an uncovered successor, and continue the traversal of the possible executions. (**REQ5.2.**)

The sequence diagram of the above-mentioned interactions between the *Simulator* and the *SimulatorListener* is shown in Figure 5.1.

5.4 Control

The simulator framework supports different *control mechanisms* (**REQ5.**) which can be achieved with different *SimulatorListener* implementations. This work details three different approaches: *interactive*, *exhaustive*, and *random*. This section gives an overview of these control methods, mainly focusing on the interaction between the *Simulator* and the *SimulatorListener*.

5.4.1 Interactive

In the case of *interactive control* (**REQ5.1.**), the goal is to let the user manually control every decision during the simulation. When the simulator notifies the listener with moreSuccessors, the listener directly interacts with the user, who can manually select from the fireable transitions based on some tracing information (e.g., line of the textual XSTS file). In this case, the simulator simulates only a single execution, i.e., there is no need to backtrack at the end of the simulation.

The sequence diagram of the interactive control is shown in Figure 5.2.

5.4.2 Exhaustive

In the case of *exhaustive control* (**REQ5.2.**), the goal is to automatically traverse every execution of the model which can be achieved with a *depth-first* approach (see Section 5.2). When the simulator notifies the listener with moreSuccessors, the listener always returns 0 as selectedSuccessorIdx, i.e., it always selects the first fireable transition. At the end of every execution, the listener makes the simulator backtrack, in order to traverse every possible execution.

The sequence diagram of the exhaustive control is shown in Figure 5.3.

5.4.3 Random

In the case of *random control* (**REQ5.3.**), the goal is to automatically traverse a randomly selected, single execution of the model. When the simulator notifies the listener with

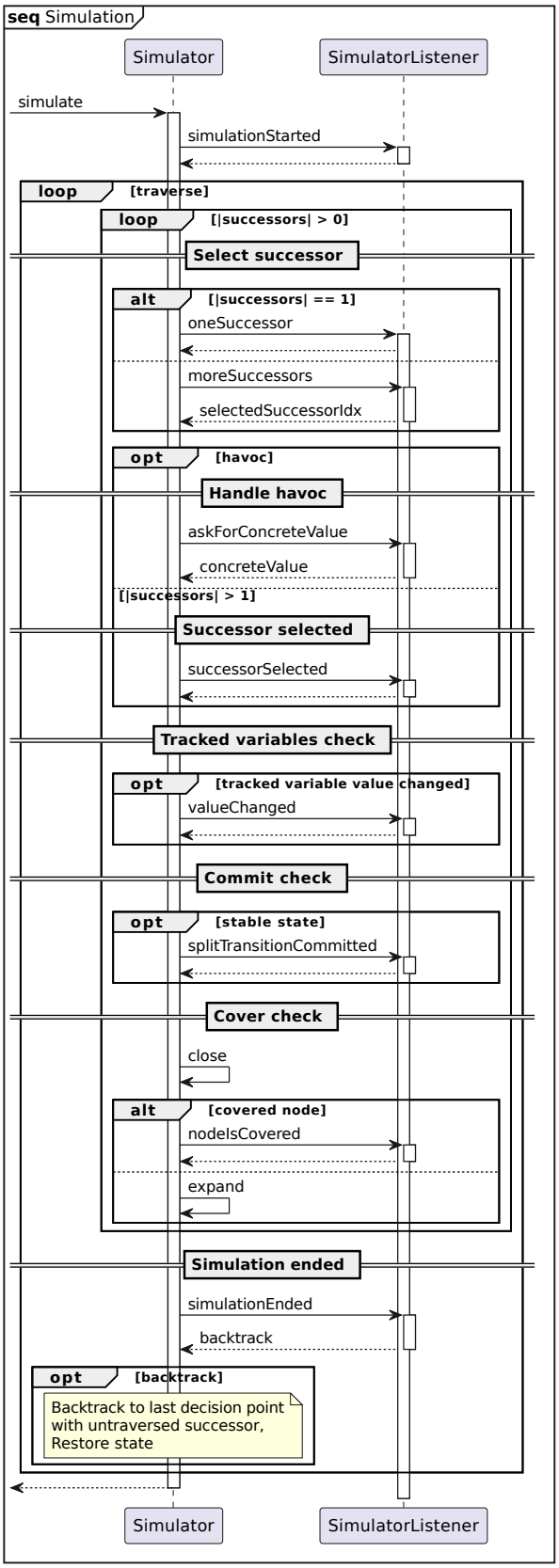

Figure 5.1: Sequence Diagram of the interactions between the Simulator and the SimulatorListener.

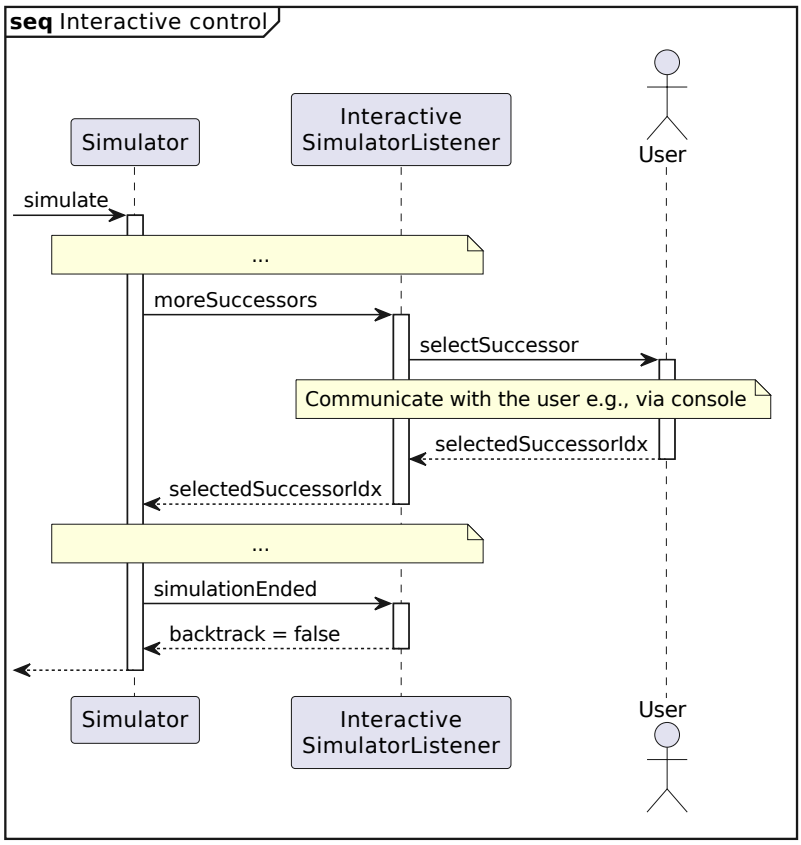

Figure 5.2: Sequence Diagram of the interactive control.

moreSuccessors, the listener works very similarly to the *interactive control*. The only difference is the calculation of the selectedSuccessorIdx: in this case, instead of explicitly asking the user, a random number is generated. In this case, the simulator simulates only a single execution, i.e., there is no need to backtrack at the end of the simulation. (For *Monte Carlo*-like sampling, several random executions are needed.)

The sequence diagram of the random control is shown in Figure 5.4.

5.5 Observation

The simulator framework supports different *observation mechanisms* (**REQ6.**) at different levels. This work details three kinds of observation: *simulation states*, *tracked variables*, and *executions*. This section gives an overview of these observation opportunities, focusing on their timing: simulation states and tracked variables are observed *during the simulation*, while executions are aggregated and graphically presented *after the simulation*.

5.5.1 During Simulation

After every step, the simulator tells the listener about the current *simulation state* (**REQ6.1.**) which contains the exact value of every variable. When the simulator changes the current simulation state, it compares the new and old values of every tracked variable. If the value of at least one tracked variable has changed, the simulator notifies the listener about these changes. (**REQ6.2.**)

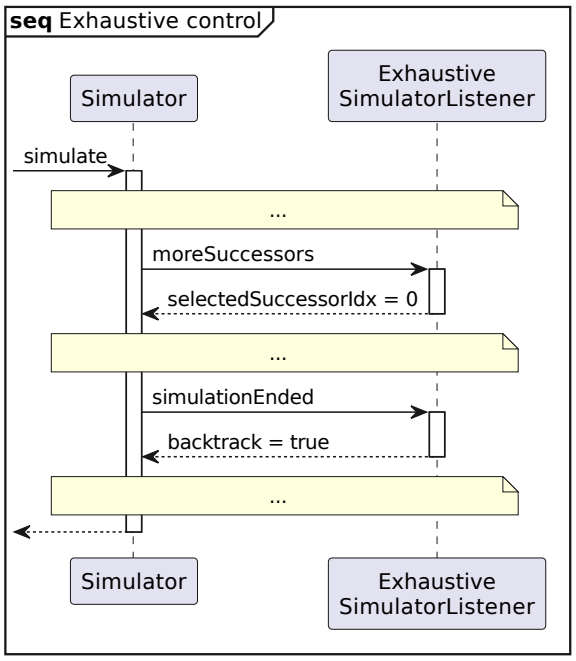

Figure 5.3: Sequence Diagram of the exhaustive control.

It is always the listener's responsibility, what to do with the information during the simulation. E.g., the listener can save the values or can show them to the user in *interactive* mode.

5.5.2 After Simulation

After simulation, the simulator aggregates the executions and presents them graphically. Currently, this feature is only available in *exhaustive* mode, where there are several executions to aggregate. I 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 straightforward, so I 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.

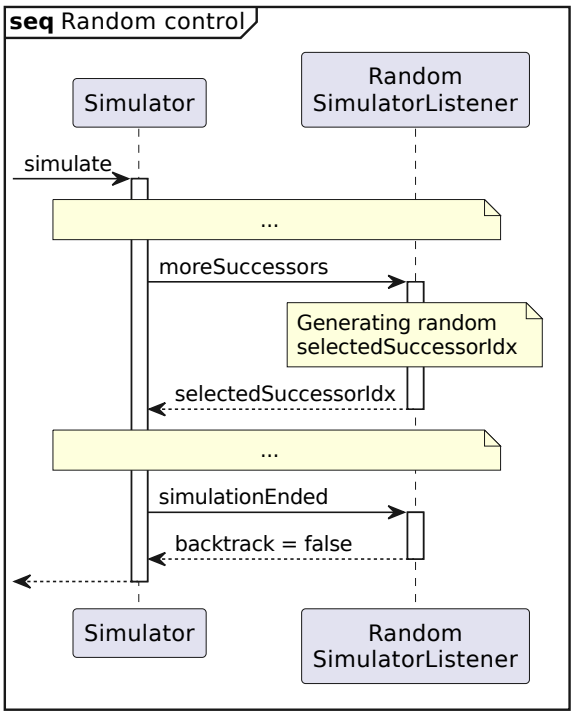

Figure 5.4: Sequence Diagram of the random control.

5.6 Implementation

I implemented the simulation framework as an extension of the Theta Model Checking Framework. Implementing the simulator in Theta has two main benefits:

1. I was able to reuse several parts of Theta which are critical in terms of preciseness. Parsing the XSTS model, and maintaining the inner representation of the state space is handled by Theta. My contribution is the implementation of the simulation layer.

2. Using the same code base for simulation and verification *guarantees* that they will always give the same results, i.e., the simulator *inherited* the preciseness of Theta.

I extended the XstsCli class, which is a simple command line interface for the model checking of XSTS models, with a new argument –simulate for defining the simulation mode. Its type is the enum SimulatorMode, with literals INTERACTIVE, EXHAUSTIVE, and RANDOM. I also added argument –simtrack of type String where the comma-separated list of tracked XSTS variable names can be defined.

If the argument –simulate is given, XstsCli instantiates the corresponding SimulatorListener implementation and the newly implemented BasicSimulator class. The BasicSimulator class follows the *builder* design pattern, i.e., its constructor is *private* and only used by the public nested (static) Builder class.

Parameters (e.g., SMT solver instance, XSTS model, SimulatorListener instance, logger, tracked variables) can be passed to the Builder instance through method calls. Then, the BasicSimulator is instantiated with the build method of the Builder class. Finally, the simulate method of the BasicSimulator is called.

Basically, the simulate method implements Algorithm 2. The simulator uses the existing ArgBuilder class to create, close, and expand the ARG. The simulator iterates through the ArgNode nodes of the ARG.

Every ArgNode contains an XstsState which contains an ExplState. An ExplState represents an explicit state and contains a Valuation object which wraps a map of variable values, i.e., contains the exact value of every variable. These Valuation objects are reused to represent the current simulation state during the interactions between the simulator and the listener.

The SimulatorListener interface serves as a common base for the different listener implementations: InteractiveSimulatorListener, ExhaustiveSimulatorListener, and RandomSimulatorListener. The simulation framework can be easily extended with further listener implementations. The interface defines the same functions as listed in Section 5.3. The arguments of these functions are SimulationState objects.

Every SimulationState contains a Valuation representing the current simulation state itself. For specific interactions, specific subclasses derive from the SimulatorState base class, containing additional interaction-specific information.

In order to visualize the resulting execution traces (see Section 5.5.2), I reused the visualization components of Theta, which can build generic graphs using the *dot* format of GraphViz[1]. I also implemented a graph simplifier module, which can simplify the generated execution traces into a more readable, compact format.

[1] https://graphviz.org/

Chapter 6

Case Study

In this chapter, I present a case study of my work, the summary of the validation of the UML PSSM standard about state machine semantics [21]. This work, co-authored by myself, was presented at the Scientific Students' Association Conference at the Budapest University of Technology and Economics in 2022.

I briefly present the *Precise Semantics of UML State Machines* (PSSM) standard and its test suite (Section 6.1) and give an overview of our approach for validating the conformance of the standard and the test suite, emphasizing the role of the simulation framework (Section 6.2). I also detail the validation through an example test case (Section 6.3).

6.1 Precise Semantics of UML State Machines

The *Unified Modeling Language* (UML) [7] 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) [8] is a follow-up specification for a subset of UML elements, refining their execution semantics. PSSM does not only define the semantics textually, but it also provides a *Test Suite* containing 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.

Every *test case* consists of a *test model*, the received *events*, and the expected *execution traces*.

The Test Suite in the standard has two main goals:

- It explicitly details the possible executions of the test models. These examples make the standard *more understandable*, so engineers can earn a deeper understanding of the modeling language semantics.
- The test models and the expected execution traces can be used for the *conformance-checking* of modeling tools. If a modeling tool yields exactly the same sets of execu-

tion traces for the test models, the tool *may* conform with the PSSM semantics. Of course, this approach does not *prove* full tool conformance, but with the appropriate selection of test models, in practice, it can provide a strong enough guarantee.

6.2 Overview

The validation workflow consists of the following steps. First, the high-level PSSM models are transformed into Gamma models preserving the semantics. The benefit of the Gamma modeling language is that it has precise formal semantics, and Gamma already provides a model transformation into XSTS models. Then, the Gamma models are transformed into XSTS models, the XSTS model is split, and the execution traces are generated with the simulator, in *exhaustive* mode.

The overview of the PSSM validation workflow is shown in Figure 6.1. The exact steps in the figure are the following.

① The textual *PSSM Semantics* defines implicit components (e.g., dispatcher) which are modeled as Gamma statecharts. These components (*Common Gamma Components*) are general PSSM components, i.e., they are reusable for every specific test model.

② The concrete *PSSM Test Model* is modeled as Gamma components (statecharts, doActivities) resulting in the *Gamma Test Model*. (Note, that step ② is manual but *systematic*, so it could be automated as a model transformation.)

③ Then, the *Common Gamma Components* and the *Gamma Test Model* are composed into one system resulting in the *Gamma Composite Test Model*.

④ The *Gamma Composite Test Model* is transformed into an *XSTS Model* using the existing Gamma-to-XSTS model transformation.

⑤ Then, splitting is applied on the *XSTS Model*, resulting in the *Split XSTS Model*.

⑥ Finally, the simulator traverses every execution trace of the *Split XSTS Model* using the *exhaustive* simulation mode, resulting in the final *Execution Traces*.

6.3 Example

In this section, I demonstrate the capabilities of the PSSM validation workflow through an example test case, Behavior 003-B [8]. I detail the test model, the expected execution trace defined in the test case, and the actual execution traces found by the *exhaustive* simulator. I also compare them and draw conclusions about the difference and the usability of the simulator.

6.3.1 Test Model

The Target State Machine of the test is shown in Figure 6.2. 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 6.3. 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.

6.3.2 Expected Trace

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

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

The specification also provides the RTC steps during execution, which is shown in Table 6.1. 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. Afterward, 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, 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 6.1: The run-to-completion steps for an execution of the Behavior 003-B [8] test case.

6.3.3 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[1] 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)`

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

- **(deadlock, final state not reached)**

The RTC steps for this trace are shown in Table 6.2. 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.

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 6.2: The run-to-completion steps for a deadlocked execution of the Behavior 003-B [8] test case.

6.3.4 Conclusion

The possible deadlock of this model was found by the *exhaustive* simulator which demonstrates the value of this approach and the usability of the simulator itself. The simulator traversed every possible execution of the model, aggregated the executions, and presented them graphically.

Figure 6.4 shows the generated execution traces by tracking only the Target log variables: $V_{T_1} = \{v_{\log_{\text{target}}}, v_{\log_{\text{doActivity}}}\}$. The simulator also found the execution trace of reaching the *deadlock* state. This trace is refined in Figure 6.5, which shows the generated execution traces by tracking the Target logs and the Dispatcher logs as well: $V_{T_2} = V_{T_1} \cup \{v_{\log_{\text{dispatcher}}}\}$.

Note, that Figure 6.4 and Figure 6.5 show the same execution traces, only with different granularity. In Figure 6.4, only the PSSM-level trace statements (modeled as Gamma log statements) are shown which also serves as an example for the back-annotation of the low-level simulation to the high-level. In Figure 6.5, the execution traces contain more details demonstrating the customizable set of tracked variables.

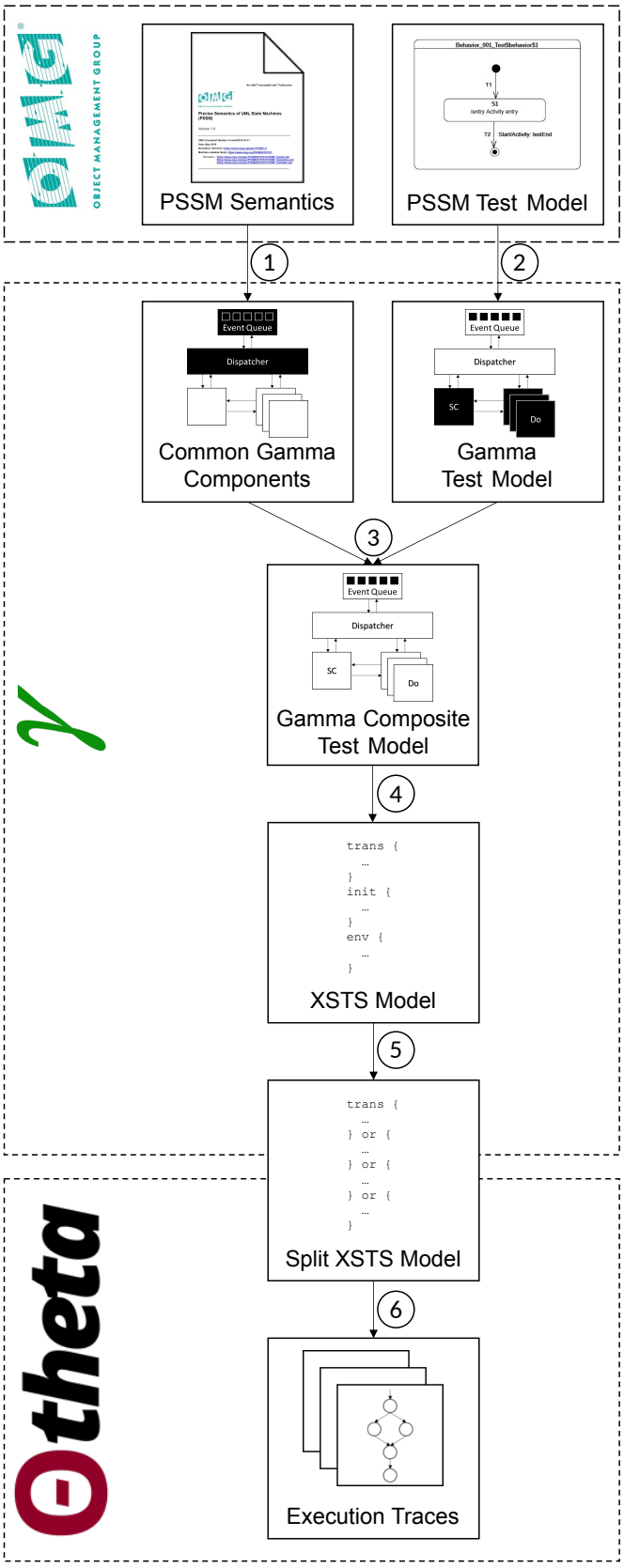

Figure 6.1: Overview of the PSSM validation workflow.

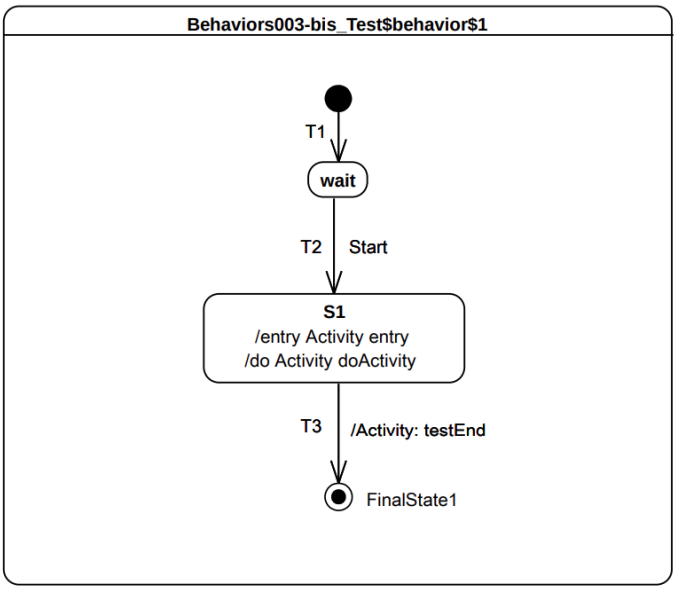

Figure 6.2: State Machine Diagram of Target component – Behavior 003-B [8] test case.

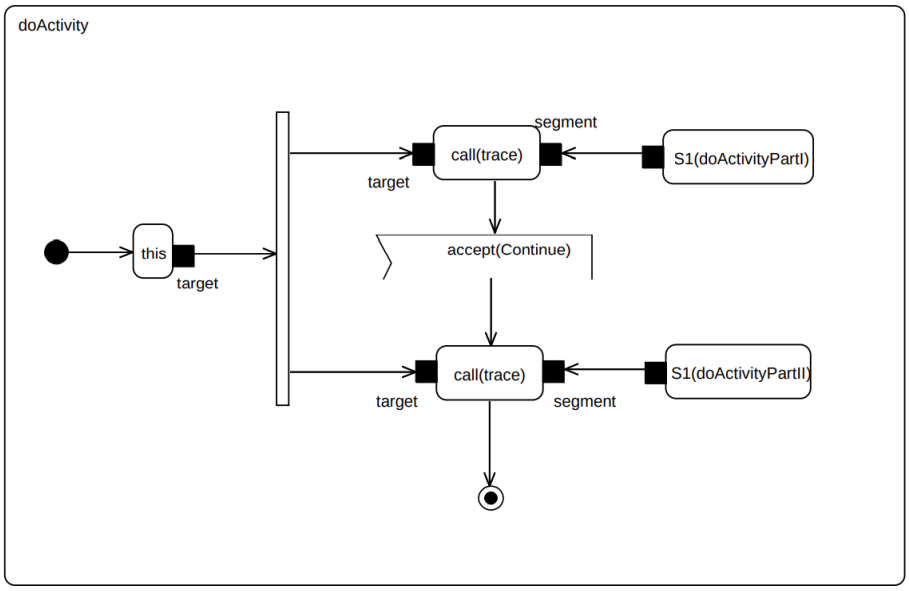

Figure 6.3: Activity Diagram of S1's doActivity – Behavior 003-B [8] test case.

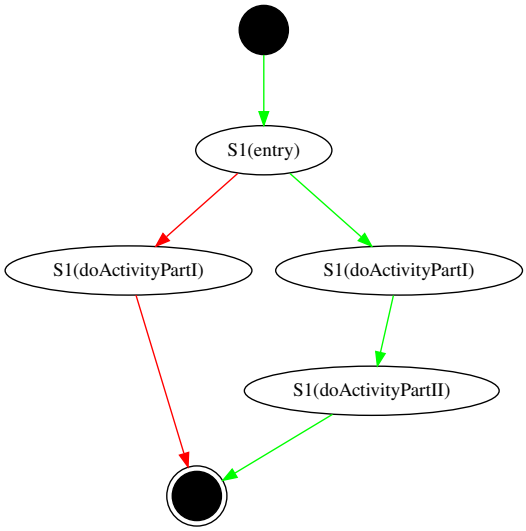

Figure 6.4: The actual execution traces of Behavior 003-B [8], found by our approach with tracking variables V_{T_1}. The expected execution trace defined by PSSM is colored green, while the execution trace reaching a deadlocked state (discussed in Section 6.3.3) is colored red [21].

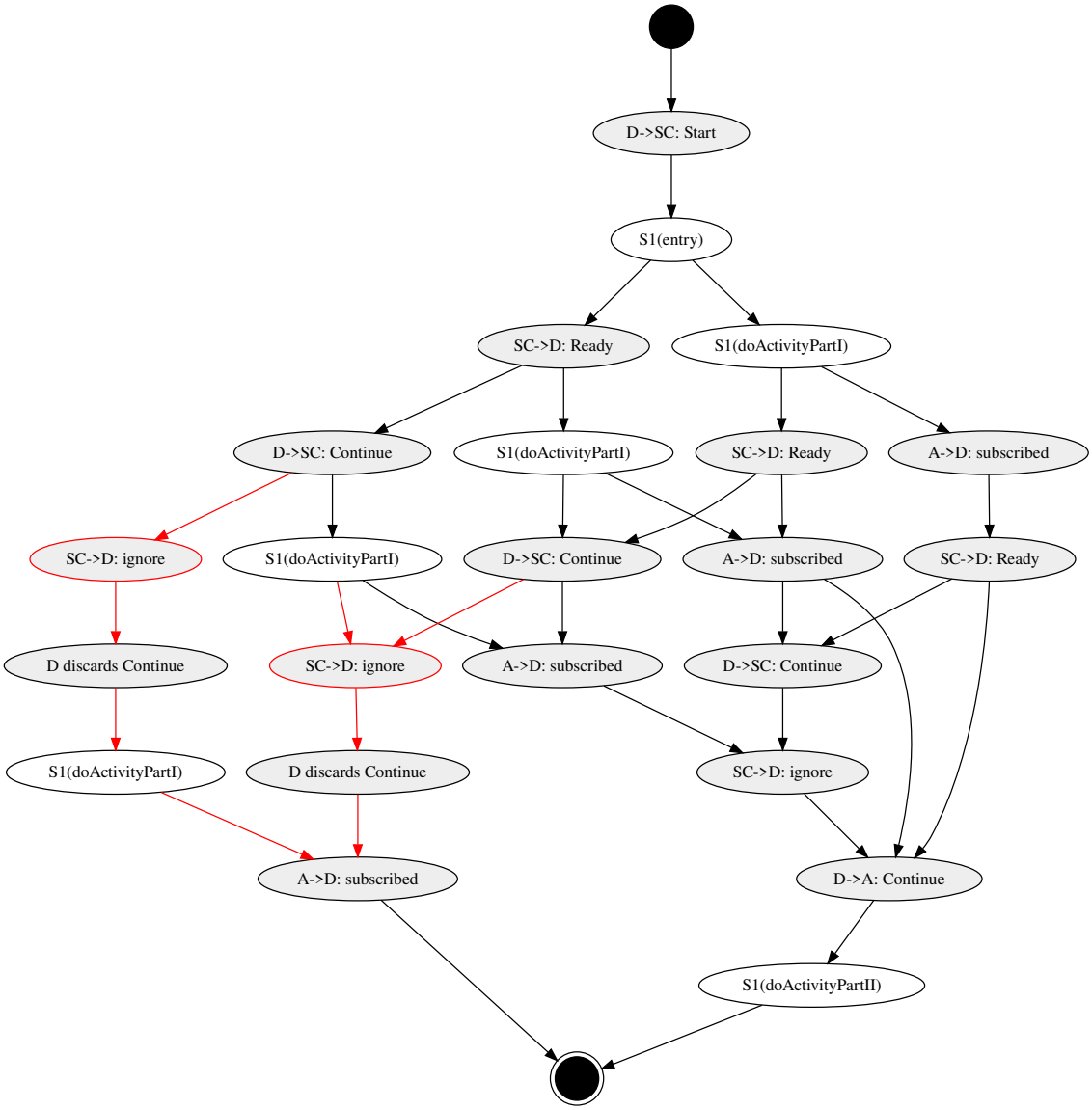

Figure 6.5: The actual execution traces of Behavior 003-B [8], found by our approach with tracking variables V_{T_2}. The nodes representing dispatcher interactions (discussed in [21]) are filled with gray, in their label D denotes the dispatcher, SC denotes the statechart, and A denotes the activity. The deadlocked execution traces (discussed in Section 6.3.3) are colored 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 [21].

Chapter 7

Conclusion

For the design of more complex systems, more complex modeling languages with higher abstraction levels are needed. Due to the increasing complexity, the need for *systematic* approaches is growing, too. In the case of behavioral models, this means the *precise* simulation of models, i.e., revealing every possible decision point during the execution, derived from the complex semantics.

In this thesis, I gave an overview of formal methods, formal models, and model simulation. For the modeling of component-based reactive systems, I presented modeling languages with different abstraction levels. These languages have precise formal semantics, as well as open-source tools supporting them.

I defined the requirements on a step-by-step controllable simulation framework for such models and presented my algorithm to make internal non-determinism external, which I implemented as an extension of the Gamma Statechart Composition Framework. I designed a simulation framework based on existing model checking infrastructure and implemented it as an extension of the Theta Model Checking Framework.

Finally, I presented the usage of my work through a case study about the validation of the UML PSSM standard using the exhaustive simulator to generate every possible execution of state machine models.

7.1 Future Work

For future improvements of the simulator, I have collected a set of requirements that could improve on existing functionalities or extend the tool with useful features.

The simulator should back-annotate every low-level (XSTS) state change to the high-level (statechart) – instead of back-annotating only the log statements. The simulator should support the tracking of general expressions over the variables – instead of supporting only individual variables. The simulator should be able to replay execution traces – supporting the saving, reloading, and replaying of simulation states/execution traces.

The user interface of the simulator could be improved, too. The simulator should be integrated into common IDEs (e.g., Eclipse or Visual Studio Code over Language Server Protocol) – making its usage more convenient and supporting widely-used debugging use cases, e.g., breakpoints.

Köszönetnyilvánítás

Számomra ez a diplomaterv a maga módján túlmutat önmagán: nemcsak a mesterképzésem lezárásának tekintem, hanem egyúttal egy mérföldkőnek is, amely az eddigi, tanulmányaim köré szerveződő életszakaszomat választja el az ezután következőtől. Éppen ezért, kissé talán rendhagyó módon, szeretném megragadni a lehetőséget, hogy itt is köszönetet mondjak mindazoknak, akik segítettek az idáig vezető úton.

Köszönöm *szüleimnek*, hogy lehetővé tették, hogy az elmúlt öt és fél évben az egyetemi tanulmányaimra koncentrálhassak, *középiskolai tanáraimnak*, hogy biztos alapokat adtak ehhez, valamint bátyáimnak, *Bencének* és *Zolinak*, hogy a legkülönbözőbb kérdéseimet is mindig igyekeztek megválaszolni. Köszönöm évfolyamtársaimnak, barátaimnak, hogy együtt jártuk végig ezt az utat: *Benedek, Dani, Domonkos, Jonatán, Józsi, Ricsi* – a segítségetek nélkül sokkal göröngyösebb lett volna.

Köszönöm a *Kritikus Rendszerek Kutatócsoport* minden tagjának, aki bármiben segített az itt töltött éveim során, különösen *Majzik Istvánnak*, hogy konzulensként segített a kezdetekben, *Micskei Zoltánnak*, hogy végig rajtam tartotta a szemét és egyengette az utamat, *Elekes Mártonnak* és *Graics Bencének*, hogy energiát nem kímélve konzultálták a TDK dolgozatunkat, valamint *Vörös Andrásnak*, hogy mindig volt egy biztató szava és hasznos észrevétele.

Külön köszönöm konzulensemnek, *Molnár Vincének*, hogy ennyi éven keresztül végigkísérte az utamat: kezdve a középiskolai szakköröktől, ahol megismertette velem a programozást, a szakdolgozatomon és a TDK dolgozatunkon keresztül egészen ennek a diplomatervnek a konzultálásáig. Lehetetlen lenne felsorolni, mennyit tanultam tőle mind szakmailag, mind emberileg, és mindig hálás leszek a feltétel nélküli támogatásáért, amit akkor nyújtott, amikor a legnagyobb szükségem volt rá.

Úgy alakult, hogy életem más területei tartogattak néhány nehézséget az elmúlt időszakban, amikor ez a diplomaterv is készült. Köszönöm barátaimnak, hogy ezekben az időkben is mellettem álltak és támogattak: *Ági, Anna, Bence, Dani, Dórik, Gergő, Hanga, Körte, Levi, Máté, Mosi, Nándi, Pali, Réka, Sajt, Szander, Viktorok, Virág, Zsófi, Zsuzsi* – nélkületek minden sokkal nehezebb lett volna.

– Vége? – kérdezte Róbert Gida.
– Ennek. De van más mesém.

A. A. Milne: *Micimackó*

Bibliography

[1] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery. ISBN 9781450373869. DOI: 10.1145/800028.808479.

[2] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11.

[3] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1–16. Springer Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1.

[4] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1–16. Springer Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1.

[5] Márton Elekes, Vince Molnár, and Zoltán Micskei. Assessing the specification of modelling language semantics: A study on UML PSSM, 2022.

[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.

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

[8] Object Management Group. Precise semantics of UML state machines (PSSM-v1.0), 2019. URL https://www.omg.org/spec/PSSM/1.0/About-PSSM.

[9] Object Management Group. System Modeling Language (SysML-v1.6), 2019. URL https://www.omg.org/spec/SysML/1.6/About-SysML.

[10] 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.

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

[12] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. DOI: 10.1109/32.588521.

[13] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[14] William Miller. *The Future of Systems Engineering: Realizing the Systems Engineering Vision 2035*. 10 2022. ISBN 9781643683386. DOI: 10.3233/ATDE220707.

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

[16] Milán Mondok. Formal verification of engineering models via extended symbolic transition systems. Bachelor's thesis, Budapest University of Technology and Economics, 2020.

[17] Nir Piterman and Amir Pnueli. Temporal logic and fair discrete systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_2.

[18] 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.

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

[20] 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.

[21] Péter Szkupien and Ármin Zavada. Formal methods for better standards: Validating the UML PSSM standard about state machine semantics. Scientific students' association report, Budapest University of Technology and Economics, 2022. URL https://tdk.bme.hu/VIK/sw8/Formalis-modszerekkel-a-jobb-szabvanyokert-Az.

[22] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, page 176–179, Vienna, Austria, 2017. FMCAD Inc., FMCAD Inc. ISBN 978-0-9835678-7-5. URL http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD17/proceedings/.