MŰEGYETEM 1782

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

Verification of Engineering Models Using a Modular Modeling Language with Configurable Semantics

Scientific Students' Association Report

Author:

Ármin Zavada

Advisor:

Bence Graics
dr. Vince Molnár

2023

Contents

Kivonat

A kritikus rendszerek, például vasúti infrastruktúra, autonóm járművek, repülőgépek vagy atomerőművek hibás működése súlyos anyagi károkon felül akár emberéleteket is veszélyeztethet; így ezen rendszerek tervezése során kiemelt fontosságú a biztonság garantálása. A biztonság biztosítására léteznek a tervezés közben is alkalmazható különböző verifikációs és validációs (V&V) technikák, azonban, ahogy növekszik a rendszerek komplexitása, úgy nehezedik az ellenőrzésük is. A mérnöki tervezői munka egyszerűsítése érdekében egy elterjedt megközelítés a modellalapú rendszertervezés, amely módszertan a dokumentumcentrikus megoldásokhoz képest központi elemmé teszi a mérnöki modellezési nyelveket. Az ezek segítségével készített tervek verifikációja és implementációja megfelelő eszközök segítségével gyorsítható és automatizálható.

Számos mérnöki modellezési nyelv érhető el (pl. UML, SysML v1 és v2, AADL), melyek hasonló, de mégis eltérő megközelítéseket alkalmaznak. Az ellenőrzés és implementáció szempontjából fontos különbségeket jelentenek a szemantikai variációk – a nyelvek hasonló szintaktikájú elemeihez gyakran kis mértékben eltérő végrehajtási szemantika társul. Hasonlóan fontos, hogy gyakran ezek a nyelvek alulspecifikált végrehajtási szemantikával rendelkeznek, melyet a rendszermérnökök és a vállalatok egyénileg (gyakran eltérően) értelmeznek. Ezen különbségek megnehezítik a nyelvek egységes kezelését, ami jelentősen lassítja a fejlett verifikációs eszközök ipari elterjedését. Ilyen verifikációs eszköz az egyetemi fejlesztésű Gamma Állapotgép Kompozíciós Keretrendszer is, amely számos modelltranszformáció segítségével képes a különböző modellvariánsok és különböző alacsonyszintű nyelvek közötti átmenetre. A keretrendszer bonyolultsága azonban a variánsok miatt egyre jobban megnehezíti a bővíthetőségét és karbantarthatóságát.

Jelen dolgozat célja egy új, moduláris szemantikával rendelkező modellezési nyelv alkalmazásának vizsgálata, amely segítségével a Gamma keretrendszerben leprogramozott, nehezen újrahasznosítható modell-transzformációk helyettesíthetők egy olyan rugalmas modellezési nyelvvel, amely alkalmas a magasszintű nyelvek struktúrájának alacsony szintű elemekkel történő modellezésére, beleértve a magasszintű elemek szemantikáját is. A nyelv egyik fő erőssége, hogy a SysMLv2 nyelvhez hasonlóan egy alap elemkészletből különféle kompozíciós technikákkal lehet összetett elemeket építeni, melyek szemantikáját vissza lehet vezetni az alap elemkészlet szemantikájára, és az így kapott elemeket újrahasználható könyvtárakba lehet szervezni. A magasszintű nyelvek leképezése így már nem modelltranszformációs, hanem modellezési feladattá válik, ahol a magasszintű nyelv szemantikai varianciáit az újrahasználható könyvtárak modellelemeinek megfelelő megválasztásával pontosíthatjuk.

Az eredményekkel lehetővé válik 1) a különböző modellezési nyelvek lehetséges szemantikáinak kipróbálása; 2) a különböző vállalatok saját konvencióinak figyelembevétele, vagyis a leképezés testreszabhatósága; 3) a formális modell optimalizálása mérésekkel alátámasztva; illetve 4) új nyelvek egyszerű bevezetése a keretrendszerbe. A javasolt módszer prototípus implementációját különböző esettanulmányokon keresztül vizsgálom, és hasonlítom össze a Gammában jelenleg is elérhető funkciókkal.

Abstract

Failure of critical systems, such as train infrastructure, autonomous vehicles, airplanes, or nuclear power plants can lead to severe economic damage or even loss of life; thus safety is a key design priority. There are various validation and verification (V&V) techniques that can be applied during systems design to ensure safety. However, as the complexity of the systems increases, so does the difficulty of their verification. Model-based systems engineering (MBSE) is a methodology aiming to reduce the complexity of engineering work, which – contrary to the document-centric approach - places models and modeling languages at the focal point of systems engineering. The verification and implementation of systems designed in these languages can be accelerated and automated using the proper tools.

Numerous engineering modeling languages exist (e.g. UML, SysML v1 and v2, AADL), that use similar yet different approaches. Important differences for verification and implementation are semantic variations - elements of languages with similar syntax often have slightly different execution semantics. It is of equal importance that these languages often have under-specified execution semantics, which are interpreted (often differently) by individual system engineers and companies. Such differences make it harder to unify the interpretation of modeling languages, thus significantly reducing the wide adoption of advanced verification tools across the industry. One such verification tool is the university-developed Gamma Statechart Composition Framework, which supports the transition from model variants to analysis languages using several model transformations. However, the increasing complexity of the framework considerably hinders its extendability and maintainability due to the number of variants by now.

This work aims to investigate the application of a new modeling language with modular semantics to replace the hard-to-reuse model transformations programmed in the Gamma framework with a flexible modeling language. The language is suitable for modeling the structure of high-level languages with low-level elements, including the semantics of high-level elements. One of the main strengths of the language is that – as in SysML v2 – it is possible to construct complex elements from a basic set of elements using various composition techniques. This allows users to map the semantics of these elements back to the semantics of the basic element set and to organize the resulting elements into reusable libraries. The mapping of high-level languages thus becomes a modeling task rather than a model transformation task, where the semantic variances of the high-level language can be refined by choosing the appropriate model elements from reusable libraries.

The results will allow 1) to compare and contrast the possible semantics of different modeling languages; 2) to take into account the conventions of different companies, i.e., to customize the mapping of models; 3) the optimization of the resulting formal model, supported by measurements; and 4) to easily introduce new languages into the framework. I investigate the prototype implementation of the proposed language through different case studies and compare it to the features currently available in the Gamma framework.

Chapter 1

Introduction

The failure of critical systems, such as those in train infrastructure, autonomous vehicles, airplanes, or nuclear power plants can result in severe economic damage or even loss of life. Thus, ensuring safety is a key priority during systems design. Various validation and verification (V&V) techniques are employed during systems design to guarantee safety. However, as systems grow in complexity, verifying them becomes increasingly challenging [26].

To simplify the engineering design work, new approaches have been adopted to help design, verify, and implement complex systems. One such methodology is Model-Based Systems Engineering [20] (MBSE), which prioritizes engineering modeling languages over document-centric solutions. The verification and implementation of designs created in such a manner can be accelerated and (at least partially) automated with the right tools, e.g., model checking [5].

In MBSE, models serve as the primary artifacts of the development process [32]. These models are expressed using various modeling languages, such as UML [14], SysML v1 [13] and SysML v2 [30], AADL, each with its specific syntax and semantics. While these languages share similar approaches, they often exhibit slight differences in execution semantics. One such difference is semantic variation - elements of languages with similar syntax often have slightly different execution semantics (e.g., top-down vs bottom-up region scheduling). Moreover, it is important to note that these languages frequently have under-specified execution semantics, leading to varied interpretations by individual systems engineers and companies [9, 33].

The increasing complexity of systems, and the trend to develop complex systems of systems (SoS) [21] necessitates the integrated use of formal methods, such as automated model checking. Systems engineers usually have no formal methods background [25], thus such solutions must apply *hidden* formal methods, i.e., the end-to-end formal verification of engineering models without user input. However, the aforementioned semantic differences between languages make it difficult to unify the processing of modeling languages, significantly reducing the wide adoption of advanced verification tools implementing hidden formal methods across the industry.

One such verification tool is the university-developed Gamma Statechart Composition Framework [27], which supports the mapping from composite statecharts to detailed analysis languages using several model transformations. Thus, the framework provides a bridge between the engineering and analysis world and allows the automatic verification of engineering models using various model checker tools. However, the increasing complexity of the framework considerably hinders its extendability and maintainability due to the number of supported execution semantics variants by now. It is difficult to customize the built-in model transformations, and it is not always possible to map engineering languages to an existing Gamma Language variant [19].

This work proposes to utilize meta-programming for formal languages, offering a straightforward method to define the execution semantics of engineering languages without the complexity of intricate built-in model transformations. To achieve this, I enhance the eXtended Symbolic Transition System [28] (XSTS), an analysis language already utilized by Gamma as an output. By extending XSTS with meta-programming capabilities, including types, inheritance, composition, transition inlining, and static recursion, engineers gain the ability to model high-level components using fundamental elements and diverse composition techniques.

Drawing inspiration from the Kernel Modeling Language [29] – the foundational language underpinning SysML v2 – I introduce a new language called Objective XSTS (OXSTS) that enables engineers to organize the modeled high-level components into reusable libraries, transforming the mapping of engineering languages into a modeling task rather than a model transformation process. This methodology allows engineers to refine the semantic nuances of engineering languages by selecting the most appropriate model elements from these reusable libraries.

Semantifyr is the new component providing the semantic mapping from OXSTS to XSTS, integrating it into the already existing Gamma ecosystem. The integration of OXSTS and Semantifyr into Gamma allows 1) the easy customization of language semantics; 2) the optimization of the resulting formal model, supported by measurements; and 3) the ability to introduce new languages into the framework with reduced effort.

My main contributions are the following.

- I define a new modular meta-programming language (OXSTS) extending XSTS, and implement a prototype transformation component (Semantifyr) for it.
- I propose a new Gamma Model Transformation Workflow using Semantifyr.
- I conduct a preliminary evaluation of OXSTS and Semantifyr using an example model.
- Finally, I draw my conclusions about the approach.

The rest of the work is structured as follows. Chapter 2 gives an overview of the theoretical background needed to understand the main contributions. In Chapter 3, an overview is given for the current and the proposed transformation workflow. Chapter 4 formulates OXSTS. Next, in Chapter 5 the OXSTS to XSTS transformation is detailed. In Chapter 6 an initial Gamma Semantic Library is implemented, mapping the implicit model elements in Gamma to OXSTS types. Next, Chapter 7 evaluates the Gamma semantic library through a case study. Finally, Chapter 8 concludes the work and lists several direct next steps.

Chapter 2

Background

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

The rest of the chapter is structured as follows. Section 2.1 overviews MBSE languages, and introduces the SysML v2 and KerML languages. In Section 2.2, the theoretical background of model checking is presented. Section 2.3 introduces the XSTS language, a formal analysis language that can be used during model checking. Next, Section 2.4 introduces the Gamma Statechart Composition Framework, which enables the formal modeling and verification of component-based reactive systems. Lastly, Section 2.5 showcases related works in the literature.

2.1 MBSE Languages

In model-based systems engineering (MBSE), models are the primary artifacts of the development process [32], which are expressed using various modeling languages, that have a language structure (abstract syntax), well-formedness constraints, exact graphical or lexical representation (concrete syntax) and an interpretation (semantics) of well-formed models. To use such models for simulation, verification, or code generation, the preciseness of the language is essential [4].

2.1.1 Systems Modeling Language v2

The Systems Modeling Language v2 (SysML v2) [30] is a general-purpose modeling language for modeling systems that is intended to facilitate an MBSE approach during system design. SysML v2 is the next generation of the widely adopted SysML [13] language, with enhanced precision, expressiveness, interoperability, and consistency.

Unlike SysML – which builds upon the Unified Modeling Language (UML) [14] – SysML v2 builds upon the new Kernel Modeling Langauge (KerML) [29]. KerML is a foundational modeling language for expressing various kinds of system models with consistent semantics.

Syntactically, KerML is divided into three layers, with each layer refining the previous one.

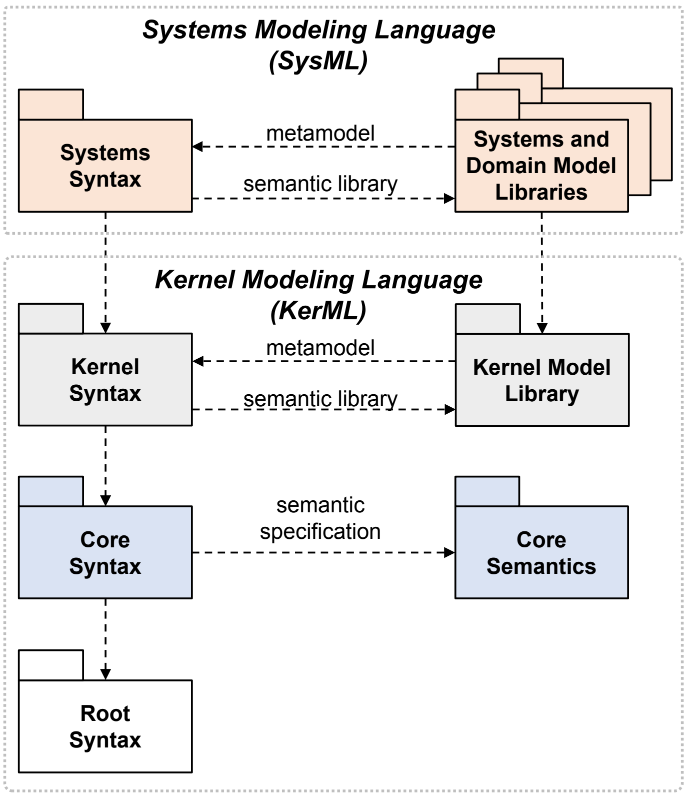

Figure 2.1: An illustration of the KerML and SysML language structure.

1. The Root Layer specifies the most general syntactic contracts for structuring models.
2. The Core Layer includes the most general constructs that have semantics based on classification.
3. The Kernel Layer provides commonly needed modeling capabilities, such as associations and behavior.
4. The Systems Layer provides high-level, systems modeling capabilities, building upon the layers below.

> “The Core Layer grounds KerML semantics by interpreting it using mathematical logic. However, additional semantics are then specified through the relationship of Kernel abstract syntax constructs to model elements in the Kernel Semantic Library, which is written in KerML itself. Models expressed in KerML thus essentially reuse elements of the Semantic Library to give them semantics. The Semantic Library models give the basic conditions for the conformance of modeled things to the model, which are then augmented in the user model as appropriate. Having a consistent specification of semantics helps people interpret models in the same way. In particular, because the Semantic Library models are expressed in the same language as user models, engineers, and tool builders can inspect the library models to formally understand what real or virtual effects are being specified by their models for the systems being modeled. More uniform model interpretation improves communication between everyone involved in modeling, including modelers and tool builders.” [29]

Figure 2.1 shows the structure of the SysML v2 and KerML languages. Since SysML v2 uses KerML as its foundation language, all the SysML v2 semantics are defined using SysML – and in turn in KerML – giving it a solid semantical foundation.

Drawing inspiration from this approach, my primary goal in this work is to introduce a similar modeling language architecture to the Gamma framework, where OXSTS (see Chapter 4) serves as the foundational language for the compact and configurable modeling of high-level language semantics. This way, the language combines the intuitiveness of SysML v2 with the formal semantics of XSTS (see Section 2.3.1)

2.2 Model Checking

Model checking is a formal verification technique used to assess the properties of systems. In essence, it evaluates whether a given formal model M satisfies a specific requirement γ. The terminology comes from formal logic, where a logical formula may have zero or more models. These models define the interpretation of symbols within the formula and the base set in a manner that renders it true. In the context of model checking, the question is whether the formal model is indeed a model of the formal requirement $M \not\models \gamma$ [5]? This question lies at the core of model checking, where rigorous analysis is conducted to determine the compliance of the system to the prescribed specifications. Model checker algorithms (see Figure 2.2), such as the ones used in UPPAAL[1] [23] or Theta[2] [34] can answer this question. If $M \not\models \gamma$, then the model checker outputs a counterexample that witnesses the violation of γ by M. This counterexample is usually returned in the form of an execution trace, allowing the engineers to understand the discrepancy with a step-by-step guide.

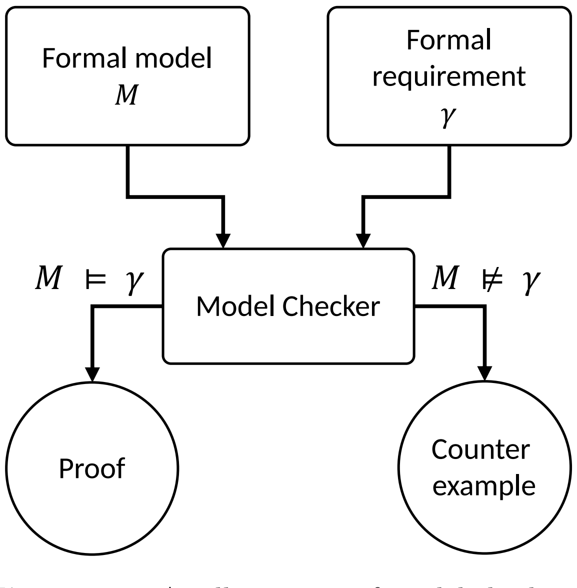

Figure 2.2: An illustration of model checking.

[1]https://uppaal.org/
[2]https://inf.mit.bme.hu/en/theta

2.3 Analysis Models

Model checker algorithms require formal analysis models. This section provides a brief overview of the analysis language used in this work. Note that the literature is much broader, and I direct the interested reader to [5, 31, 22, 17, 1, 3, 15, 18]

2.3.1 Extended Symbolic Transition System (XSTS)

Extended Symbolic Transition System (XSTS) [28] is an extension of Symbolic Transition System [18] (STS), providing an easier-to-use language for the specification of engineering models with formal semantics.

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

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

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

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

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

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

- *Assignments*: An assignment of form $v := \varphi$ with $v \in V$ and φ as an expression of the same type D_v means that φ is assigned to v in the successor state s' and all other variables keep their value. Formally, $s'(v) = \varphi \wedge s'(v') = s(v')$ for every $v' \neq v \in V$, while $|s'| = \frac{|s|}{|s(v)|}$.

- *Assumptions*: An assumption of form $[\psi]$ with ψ as a Boolean expression over the variables $(var(\psi) \subseteq V)$ checks condition ψ without modifying any variable and can only be executed if ψ evaluates to *true* over the current state s, in which case the successor state is $s' = s$, and $|s'| = |s|$ – otherwise the set of successor states is the empty set $\emptyset$, and $|s'| = 0$.

- *Havocs*: A havoc of form *havoc*(v) with $v \in V$ means a non-deterministic assignment to variable v, i.e., after execution, the value of v can be anything from D_v and all other variables keep their value. Formally, $s'(v) = \top \wedge s'(v') = s(v')$ for every $v' \neq v \in V$. Therefore, c'_i will be $|s'| = |D_v| * |s|$.

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

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.

2.3.2 Theta Model Checking Framework

Theta[3] [34] is a generic, modular and configurable model checking framework developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics, aiming to support the design and evaluation of abstraction refinement-based algorithms [16] for the reachability analysis of various formalisms. Theta is capable of processing – among many others – XSTS models.

2.4 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework[4] [27] is an integrated tool to support the design, verification, and validation of, as well as code generation for component-based reactive systems. The behavior of each atomic component is captured by a statechart while assembling the system from components is driven by a composition language. Gamma supports several composition semantics, allowing the user to model systems with various (potentially mixed) execution and interaction semantics.

Gamma integrates with various model checker tools, including Theta, introduced in Section 2.3.2.

2.4.1 Gamma Behavioral Languages

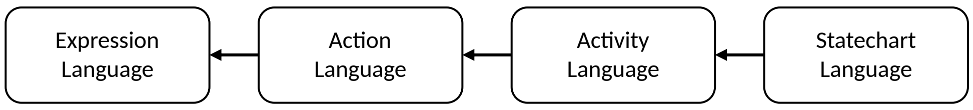

Figure 2.3: The language structure of the Gamma Framework.

Figure 2.3 displays the language structure of the behavioral Gamma languages.

To model various system *behaviors*, Gamma defines several formal languages, of which the *Gamma Expression Language (GEL)* and *Gamma Action Language (GAL)* [35] serve as the foundation. GEL and GAL together define *variables*, *types*, and *expressions* accessing and combining them using *arithmetical* and *logical* expressions. GAL builds on these constructs by providing simple *atomic actions* over variables in a reusable fashion.

A previous work [37] proposed the *Gamma AcTivity Language* (GATL) which is an extension of the Gamma Action Language, providing control- and data-flow semantics for modeling concurrent systems. GATL provides *simple* and *composite* actions, *fork-join* and *decision-merge* control nodes, and *action pins* for data flow modeling.

The most basic building blocks of Gamma components are atomic components, of which Gamma currently supports statecharts with the *Gamma Statechart Language* (GSL) [11].

[3]`https://inf.mit.bme.hu/en/theta`
[4]`https://inf.mit.bme.hu/en/gamma`

Statechart formal semantics provide simple and composite states, entry-exit and doActivity behaviors, orthogonal regions, and transitions with effects.

A previous work [33] further extended GATL with a new Activity Component, that can be used just like the Statechart components, with the difference that they adhere to activity semantics.

Listing 2.1 and Listing 2.2 present example statecharts. Both statecharts have two states: Idle and Operational, and both statecharts' execution starts in the Idle state. The Leader has two ports, *control* and *start*. When the *fire* event comes in through the *control* port, the Leader transitions to the Operational state and sends the *start* event through its *start* port. The Leader statechart transitions back to Idle upon a *stop* event. The Worker statechart transitions to Operational upon the *start* event.

```
statechart Leader [
    port control : requires Control
    port start : provides Start
] {
    region Main {
        initial Entry
        state Idle
        state Operational
    }

    transition from Entry to Idle
    transition from Idle to Operational
        when control.fire /
        raise start.start;
    transition from Operational to Idle
        when control.stop
}
```

Listing 2.1: The leader statechart.

```
statechart Worker [
    port start : requires Start
] {
    region Main {
        initial Entry
        state Idle
        state Operational
    }
    transition from Entry to Idle
    transition from Idle to Operational
        when start.start
}
```

Listing 2.2: The worker statechart.

2.4.2 Gamma Composition Semantics

As shown in Figure 2.3, the *Gamma Composition Language* uses the Gamma Statechart Language as its main behavioral language for defining atomic components. Indeed, atomic component behavior can be modeled using statecharts or activities; however, Gamma also provides a powerful composition language to combine different kinds of components to model various interaction and execution semantics. This section provides a detailed overview of the composition semantics based on [12].

	Atomic	**Composite**
Synchronous	Statechart	Synchronous composite component
	Activity Component	Cascade composite component
Asynchronous	Asynchronous adapter	Scheduled asynchronous composite component
		Asynchronous composite component

Table 2.1: The various components Gamma supports, grouped in atomic-composite and synchronous and asynchronous categories.

Components serve as types of component instances. They may be *atomic* or *composite*, *synchronous* or *asynchronous*. A component can have zero or more ports, which serve as the only point of interaction between components. This ensures that external dependencies and interactions are explicitly modeled, leading to a fully encapsulated behavior. Table 2.1 summarizes the various components Gamma supports.

Atomic components can be considered black boxes, with

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

New kinds of atomic components introduced into the framework must follow these rules.

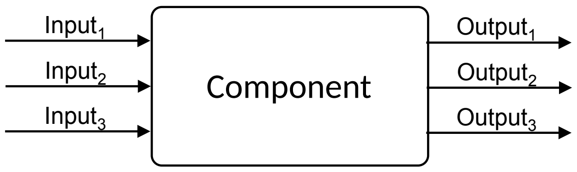

Figure 2.4: Abstract diagram of atomic components.

Synchronous Components

The execution of synchronous components is scheduled by a scheduler, which invokes the execution of the component using the *cycle*[5] input. The execution of atomic components follows a turn-based semantic, where a turn is called a *cycle*. In a cycle, the component processes its incoming signals and produces output signals by its internal state. Output signals are present for a single execution cycle only, meaning the signal disappears after one cycle (in case it is not raised again in the next cycle). An illustration of an abstract atomic component is shown in Figure 2.4, depicting a component with a set of input and output signals.

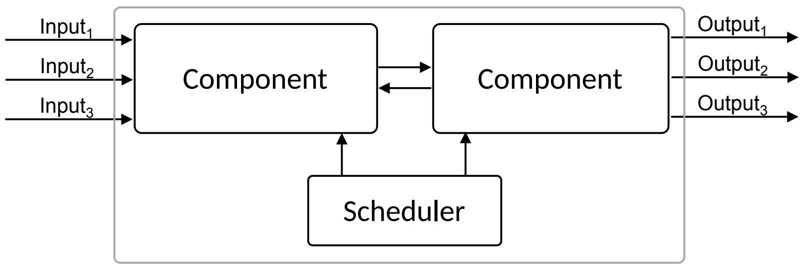

Figure 2.5: Structure of a synchronous composite component.

Synchronous *composite* components are defined by their internal components and their connections, i.e., channels and port bindings. Composite components may contain one or more *channels*, which connect internal components. The composite component's ports may also be *bound* to an internal component's port, exposing it to the environment. The behavior of the component is defined by the scheduler, which executes the internal components. This execution may be in sync (channels run after all components) or cascade (specific channels run after the corresponding component) order. Figure 2.5 depicts a generic composite component and its internal structure.

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

Asynchronous Components

Asynchronous behavior is supported by injecting buffers between the components. In Gamma, *event queues* can be used to achieve the delayed processing of incoming events in a component. Event queues may contain multiple events and have priorities over each other, affecting *when* a specific inner component is scheduled. To use atomic components in asynchronous contexts, an *asynchronous adapter* must be used, which wraps the component, making it compatible with asynchronous systems. Figure 2.6 depicts an asynchronous adapter component with two separate event queues. *Queue$_{1\text{-}2}$* has a priority of 1, while *Queue$_3$* has a priority of 2 – thus events from *Input$_1$* and *Input$_2$* will be processed before events from *Input$_3$*.

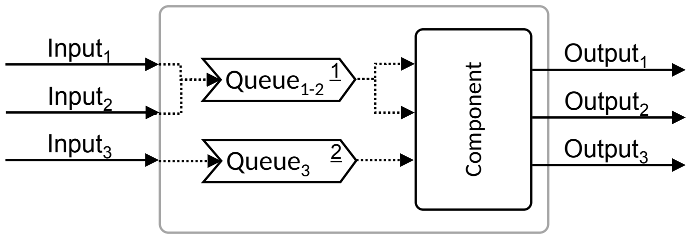

Figure 2.6: Diagram of an asynchronous adapter component.

Asynchronous composite components compose other asynchronous components, connecting them with channels – similarly to synchronous components. Figure 2.7 depicts an asynchronous composite component with two internal components and queues. Asynchronous components are inherently nondeterministic, meaning there is no guarantee on the execution time and frequency of the components, only on the ordering between the processing of the events – events in higher priority queues will be processed first, in the order of their arrival.

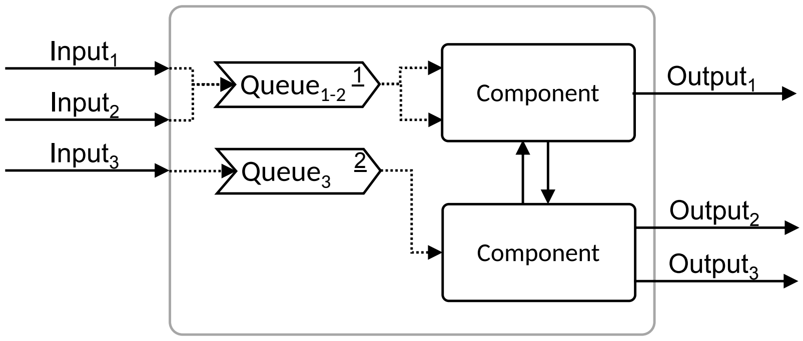

Figure 2.7: Structure of an asynchronous composite component.

Listing 2.3 composes the statecharts presented in Listing 2.1 and Listing 2.2. By connecting the two statecharts with a channel, the two now communicate. Since the System is a synchronous component, the two statecharts pass events to each other using the synchronous composition semantics.

```
sync System [
    port control : requires Control
] {
    // Instantiating the leader and worker statecharts
    component leader : Leader
    component worker : Worker

    // binding input port
    bind control -> leader.control
    // and connecting the leader to the worker
    channel [ leader.start ] -o)- [ worker.start ]
}
```

Listing 2.3: Gamma system of the Leader and Worker statecharts presented in Listing 2.1 and Listing 2.2.

2.5 Related Work

This section summarizes some of the related works in the literature.

Elekes et al. in [9] investigate the assessment of modeling language specifications in regards to (i) whether they contain errors, contradictions, or ambiguities, (ii) how suitable they are for assessing the correctness of related modeling tools, and (iii) how helpful they are for professionals to understand the language. As a result, they have pinpointed several significant errors in the PSSM Test Suite specification and test traces, resulting in unclear semantics. This work showcases the imprecise execution semantics of high-level engineering languages and the need for precise modeling languages.

Ma et al. in [25] (i) conduct an extensive literature review on existing domain-specific modeling methods (DSMM) engineering approaches, (ii) provide a detailed description of validation and verification for each phase of DSMM engineering, (iii) and a road-map encompassing the desiderata for further advances in V&V in DSSM engineering. The authors advocate for the use of formal methods in DSMM engineering, however, also acknowledge that the field of formal methods is often not part of conceptual modeling courses.

Cuccure et al. in [6] propose a template-based notation enabling semantic variation points in modeling languages to be clearly and explicitly identified within the metamodel, using template parameter definitions. Using their approach, semantic variation points can be fixed by parameter binding at both model and metamodel levels.

In a previous work [37], I defined a new Gamma behavioral language extending the framework with activity semantics. Using the Gamma Activity Language, a frequently used modeling technique, called “doActivity” can be implemented and formally verified.

In a previous work [33], Péter Szkupien and I extended the Gamma Activity Language with composition semantics, resulting in the Activity component extension. Activity components can be connected to classic Gamma statecharts using the built-in port semantics. Using this approach, activity components and statecharts can be composed using any composition semantics supported by Gamma. We defined the precise execution semantics of UML State Machines using the introduced new formalism.

Yannis Lilis and Anthony Savidis conducted an extensive survey of existing meta-programming languages in [24]. The survey classified the meta-programming languages into several categories, including the main metaprogramming model employed by the language, their phase of evaluation, the location of the program’s source, and finally, the relation between the meta-language and the object language. This paper served as the main source of insights in the context of meta-programming during the design phase of OXSTS.

Ebner et al. in [8] introduce a new meta-programming framework used by Lean [7], an interactive theorem prover based on dependent type theory. They added the meta keyword to the traditional logic language, using a similar approach as OXSTS does: with the use of meta keywords, frequently used patterns can be simplified and reused.

Chapter 3

Overview

The motivation for this work arises from the challenges faced by existing verification tools, particularly the Gamma Statechart Composition Framework. While Gamma provides a bridge between engineering and analysis domains, enabling automatic verification of engineering models, its complexity hampers extendability and maintainability. Customization of built-in model transformations and mapping diverse engineering languages to existing Gamma variants proves difficult. This chapter provides an overview of the current and envisioned model transformation workflow.

The chapter is structured as follows. Section 3.1 provides an overview of the end-to-end model transformation workflow, showcasing Gamma's already existing features. Section 3.2 introduces the details of the proposed new workflow. Finally, Section 3.3 defines several requirements for the proposed OXSTS language.

3.1 Gamma Model Transformation Workflow

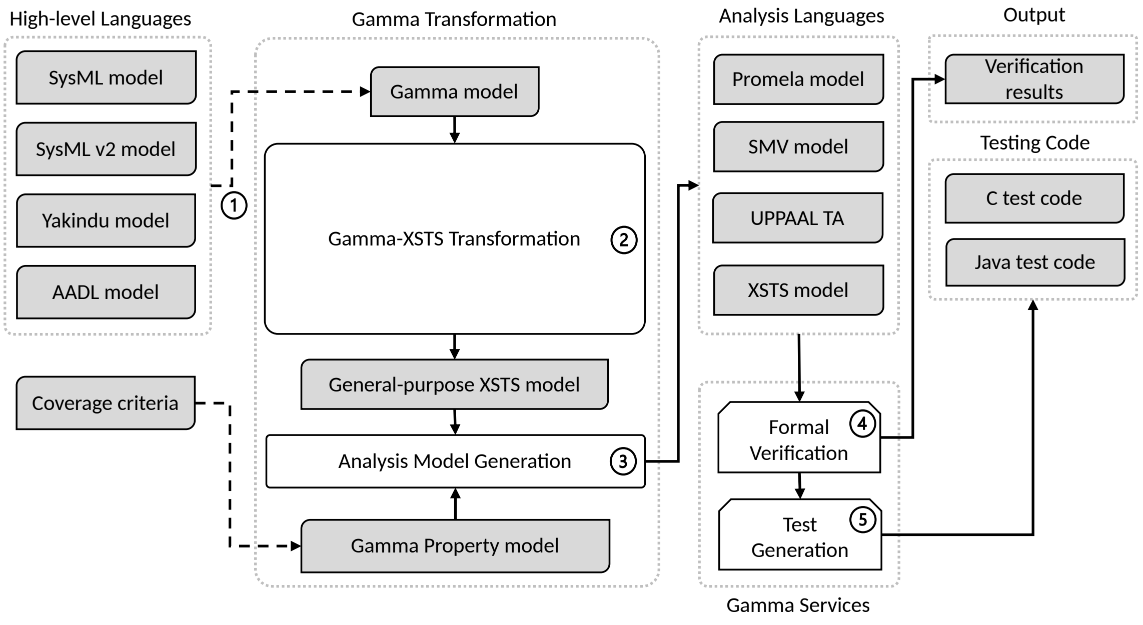

Figure 3.1: The current end-to-end model transformation workflow of the Gamma framework.

Figure 3.1 shows the current Gamma model transformation workflow. Out of the box, Gamma supports Yakindu and SysML v2, however, previous works have used Gamma to verify SysML models as well [19]. ① To process such languages, the models must be transformed into valid Gamma models. ② Afterwards, the Gamma models are transformed into a general-purpose XSTS model. By specifying coverage criteria, users can specify properties for the backend model checkers to verify. ③ The general-purpose XSTS model is transformed into the various back-end analysis languages. At this point, the model is optimized to the concrete verification properties. ④ Using these models, Gamma can seamlessly utilize various model checkers, providing a fully hidden formal verification experience for the users. The verification results are then mapped back into Gamma representations. ⑤ The framework is also able to generate concrete test cases from the verification results, which may be used as a conformance test suite for the final implementation of the system.

3.2 Workflow with Semantifyr

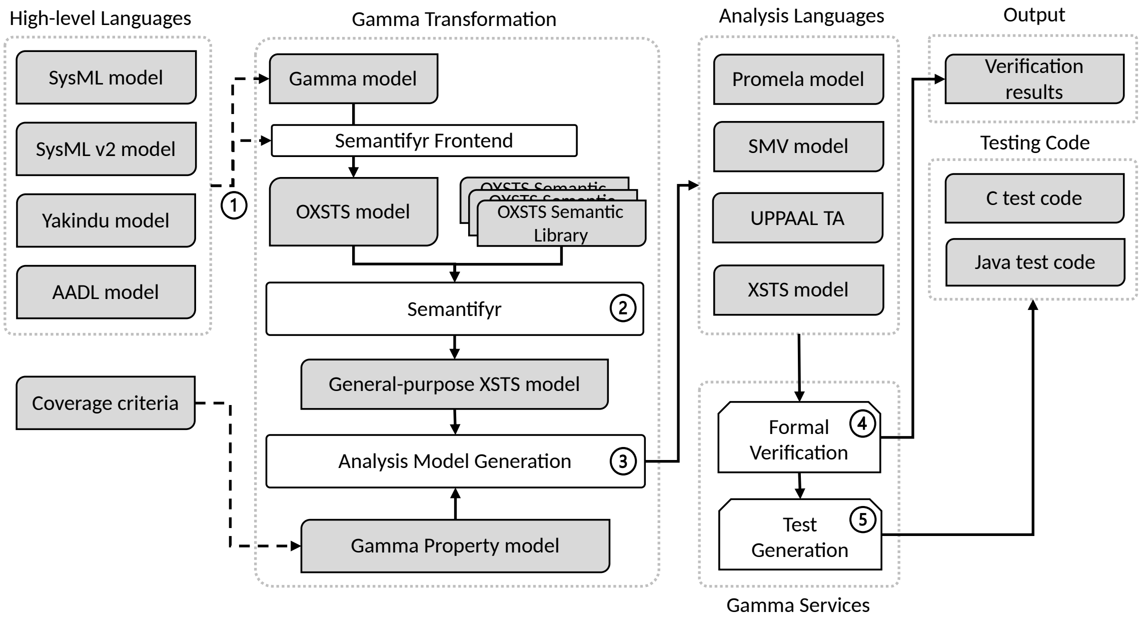

Figure 3.2: The proposed end-to-end model transformation workflow using the new Semantifyr component.

In the refined workflow depicted in Figure 3.2, the traditional Gamma-XSTS Transformer component is replaced with the new Semantifyr. ① Semantifyr operates with OXSTS models as input, requiring the mapping of frontend languages to OXSTS. However, by creating a dedicated Semantic Library for each language, the mapping simplifies into a straightforward 1-to-1 transformation from the input language to OXSTS. The Semantifyr Frontend is not yet implemented. The Frontend would be capable of the automatic mapping of models specified in Gamma or other frontend languages to OXSTS. ② Subsequently, the OXSTS models are transformed into XSTS according to the specifications of the referenced Semantic Libraries. This direct mapping to OXSTS eliminates the need to map all language semantics to the languages of the Gamma framework. Instead, these semantics can be precisely defined using a tailored Semantic Library, streamlining the entire mapping process. Additionally, this approach facilitates the creation of semantic variants without altering the fundamental transformation implementation.

3.3 OXSTS Language Requirements

The Objective XSTS language must adhere to some high-level requirements to ensure the previously introduced workflow can be achieved. The requirements for OXSTS can be outlined as follows.

1. **Formal Semantics**: OXSTS must have well-defined and formal semantics, ensuring precise interpretation and execution of engineering models. This formal foundation is crucial for accurate verification and validation processes.
2. **Familiarity to Engineers**: OXSTS must be designed with an intuitive syntax and concepts familiar to systems engineers.
3. **Modularity**: OXSTS must be inherently modular, allowing engineers to reuse Semantic Libraries across languages with small semantic variations.
4. **Expressiveness**: OXSTS must be expressive enough to capture the intricacies of high-level engineering concepts. It should support a wide range of modeling constructs and composition techniques, enabling engineers to represent diverse system behaviors accurately.

By meeting these requirements, OXSTS would provide engineers with a powerful and user-friendly formal language, enhancing the efficiency and effectiveness of the formal verification process for complex engineering systems, and ultimately bringing the power of hidden formal methods closer to systems engineers.

Chapter 4

Meta-programming for XSTS

Objective XSTS is a new language implementing several meta-programming features known across the industry, built on top of the XSTS formal analysis language. This chapter provides a walkthrough of the design, textual representation, and features of the language.

The chapter is structured as follows. Section 4.1 introduces the principles guiding the design of the language. In Section 4.2, an overview is presented of the language. Finally, Section 4.3 gives a guided tour of the language features.

4.1 Design Principles

During language design, the main focus was to satisfy the requirements specified in Section 3.3, to provide a powerful and user-friendly formal language. The following list discusses how specific requirements are satisfied with the language.

1. **Formal Semantics**: One of the foundational principles driving the design of OXSTS is the assurance of formal semantics. Recognizing the critical importance of formal verification in engineering processes, OXSTS is built upon the solid formal semantics of XSTS. Thus, OXSTS inherits the formal semantics of the XSTS language.

2. **Familiarity to Engineers**: By incorporating already familiar constructs such as types, inheritance, and composition, the language is tailored to be easily understandable and usable for systems engineers and verification tool engineers alike. Another approach would have been to use KerML or SysML v2 as the foundational language, thus providing familiarity to KerML users. However, it was ultimately decided against. Since KerML does not yet have formal execution semantics, it would have been difficult to integrate with XSTS. Ultimately, KerML served as inspiration during the design of the language, various constructs have been incorporated to make OXSTS simple to integrate into KerML in the future.

3. **Modularity**: OXSTS emphasizes modularity as a key design principle. The language supports modularity through features like composition and transition inlining, allowing engineers to break down complex systems into manageable and reusable components. With inheritance, engineers can extend already existing semantics with new variants, just by “overriding” specific transition definitions in sub-types.

4. **Expressiveness**: OXSTS inherits the expressiveness of XSTS, and extends it with familiar and simple-to-use features. Since Gamma already uses XSTS as analysis language, any Gamma model can be defined in XSTS, and thus in OXSTS as well.

Extension of XSTS deliberately simplifies existing modeling patterns using simple-to-use language features, while keeping the well-defined execution semantics of XSTS. The long-term aim is to integrate OXSTS with KerML, however, that is beyond the scope of this work.

4.2 Language Overview

The motivation behind OXSTS arises from the need for a modeling language that seamlessly combines the robustness of concrete XSTS constructs with the flexibility of compile-time features. This combination allows verification tool engineers the simple customization of already existing language semantics, and the addition of new languages to their tools. OXSTS addresses this need by offering a structured approach to semantical modeling, allowing for the replication of common patterns and the creation of adaptable models. By incorporating compile-time constructs like types, inheritance, and transition inlining, OXSTS ensures that models are not only expressive and compact but also easily customizable.

OXSTS is designed with two fundamental aspects. The first aspect delves into the realm of concrete XSTS constructs, encompassing variables, transitions, and enums. These elements form the backbone of the language, providing the essential building blocks for modeling intricate systems.

The second aspect of OXSTS focuses on compile-time constructs, introducing concepts such as types, transition inlining, features, inheritance, and feature-typed variables. These meta-programming constructs serve a dual purpose. They enhance the clarity and structure of the semantic model and also offer powerful simplification techniques. Additionally, these constructs create extension points, enabling the semantic model to be easily customizable and extensible.

4.3 Language Features

OXSTS has many language features. This section provides a detailed walkthrough of the meta-programming constructs OXSTS brings to the world of XSTS, such as transition inlining, types, features and composition, static recursion, and polymorphism.

4.3.1 Choice-else

In XSTS, choices represent non-deterministic behavior, in which exactly one branch is chosen for execution. If none of the branches may be executed (failing assumptions, see Section 2.3.1), then the whole choice operation fails to execute, which prohibits the whole operation structure from executing. For this reason, choices often have a “default” branch, whose assumptions only evaluate to true *iff* all others evaluate to false. In OXSTS, choices are extended with else branches. This serves as an easy-to-use syntax sugar and is mapped to the classic form. Listing 4.3.1 depicts an example of the classic and new choice-else syntax.

```
choice {
    assume (x != 10)
    // ...
} or {
    assume (y == 20)
    // ...
} or {
    // ...
} or {
    assume (!((x != 10) || (y == 20) || ...))
    // Some operation
}
```

```
choice {
    assume (x != 10)
    // ...
} or {
    assume (y == 20)
    // ...
} or {
    // ...
} else {
    // Some operation
}
```

Listing 4.3.1: Choice-else using classic XSTS (left) and the new else syntax (right).

4.3.2 Target

The entry points of OXSTS models are **Target** definitions. Target definitions define the "universe" of the model, grounding the model to a specific configuration. A Target is equivalent to an XSTS model, containing *variables*, *transitions* and *properties*.[1] An OXSTS model may contain multiple target definitions, in which case they represent different configurations of the model.

Listing 4.3.2 depicts a simple Target (left) and its corresponding XSTS model. Note that, contrary to XSTS, OXSTS does not define *env* transitions. This might seem to reduce its expressiveness, however, it does not: by placing all environmental transitions to the front of the internal transition, the same behavior is captured without the need for additional complexity.

```
enum TimeUnit { // enumeration definition
    Seconds, Minutes, Hours
}
target Mission {
    var unit : TimeUnit // variables
    var x : Integer := 1 // initial value
    var isZero : Boolean := false
    init { // init transition
        unit := TimeUnit::Seconds
    }
    tran { // main transition
        unit := TimeUnit::Minutes
        havoc (x)
        isZero := x = 0
    }
    prop { // model invariant
        isZero = false
    }
}
```

```
type TimeUnit : {
    Seconds, Minutes, Hours
}
// variables get a prefix for
    differentiation
var __Mission__unit : TimeUnit;
var __Mission__x : integer = 1
var __Mission__isZero : boolean = false
init {
    __Mission__unit := Seconds;
}
trans {
    __Mission__unit := Minutes;
    havoc __Mission__x;
    __Mission__isZero := __Mission__x == 0
}
// env transition is always empty
env {}
prop {
    ((__Mission__isZero) == false)
}
```

Listing 4.3.2: Basic Target definition on the left, with its equivalent XSTS representation on the right, transformed by Semantifyr.

[1]Properties specify invariants for XSTS models, used as reachability properties by Theta.

4.3.3 Transition Inlining

Transition inlining is a meta-programming feature of OXSTS. XSTS models often contain the same lines of operations over and over again. With inlining, they can be extracted into reusable forms. Transitions may have parameters, which can be *bound* at the inline site by specifying them between the parentheses.

The semantics of inlining can be regarded as refactoring steps applied again and again over the model. Listing 4.3.3 shows an example of transition inlining and its corresponding refactored form.

```
target Mission {
    var x : Integer := 1
    var isZero : Boolean := false
    tran setIsZero() {
        isZero := x = 0
    }
    init {
        x := 2
        inline setIsZero()
    }
    tran {
        havoc (x)
        inline setIsZero()
    }
    prop {
        isZero = false
    }
}
```

```
target Mission {
    var x : Integer := 1
    var isZero : Boolean := false
    init {
        x := 2
        // inlined transition
        isZero := x = 0
    }
    tran {
        havoc (x)
        // inlined transition
        isZero := x = 0
    }
    prop {
        isZero = false
    }
}
```

Listing 4.3.3: A basic Target definition with transition inlining (left) and its corresponding refactored version (right).

The language also supports conditional inlining with *inline if* operations. If the specified compile-time evaluable expression evaluates to true the *body* is inlined, otherwise the *else* is (or nothing, if there is no else). Listing 4.3.4 depicts a simple example of a conditional inlining. Note, that with more advanced features (see Section 4.3.7) the condition can be non-constant as well.

```
// constant specification
const DO_SUBTRACTION: Boolean := true
target Mission {
    var x : Integer := 1
    tran {
        inline if (DO_SUBTRACTION) {
            x := x - 1
        } else {
            x := x + 1
        }
    }
}
```

```
target Mission {
    var x : Integer := 1
    tran {
        x := x - 1
    }
}
```

Listing 4.3.4: A simple inline-if with a compile-time evaluable (in this case constant) expression.

4.3.4 Types

In case the same variables and transitions must be repeated, Types can be used to reduce code duplication. Types can be instantiated, which means their variables and transitions can be accessed from the instantiation site. Listing 4.3.5 shows a simple instantiated Type and its corresponding refactored version.

```
type Container {
    var x : Integer
    var isZero : Boolean
    init { // init transition
        x := 2
        isZero := x = 0
    }
    tran { // main transition
        havoc (x)
        isZero := x = 0
    }
}
target Mission {
    // instantiating type Container
    instance containerA : Container
    instance containerB : Container
    init {
        // init transition reference
        inline containerA.init()
        inline containerB.init()
    }
    tran {
        // main transition reference
        inline containerA.main()
        inline containerB.main()
    }
    prop {
        containerA.isZero = false &&
        containerB.isZero = false
    }
}
```

```
target Mission {
    // variables transformed from the instance
    var containerA__x : Integer
    var containerA__isZero : Boolean

    var containerB__x : Integer
    var containerB__isZero : Boolean

    init {
        // inlined init transitions
        containerA__x := 2
        containerA__isZero := containerA__x = 0
        containerB__x := 2
        containerB__isZero := containerB__x = 0
    }

    tran {
        // inlined main transitions
        havoc (containerA__x)
        containerA__isZero := containerA__x = 0
        havoc (containerB__x)
        containerB__isZero := containerB__x = 0
    }

    prop {
        containerA__isZero = false &&
        containerB__isZero = false
    }
}
```

Listing 4.3.5: Simple Type instantiation in a Target definition (left) and its refactored version (right).

4.3.5 Features and Composition

To facilitate various composition techniques, types may have several kinds of relations. **Features** specify composition relations between instances. **References** are special features without composition. **Instance** relations are **Features** that must contain exactly one instance, and thus are used to force the creation of concrete instances. Features specify the types that they relate to, and may also have *multiplicities*, specifying the cardinality of their instances. The following multiplicities are supported.

- Optional `[0..1]`
- One `[1..1]`
- Some `[0..*]`

A classic composition technique [30] is to specify *subsetting* relations between features. If Feature A is subsetted by Feature B, then any instance contained (or referenced) in Feature B is also included in Feature A. Subsetting is marked with the `:>` symbol.

```
type Region {
    // composition relation "States"
    feature states : State[0..*]
    // entryStates are also states
    feature entryState :> states : State[1..1]
}
type State {
    // reference to parent region
    reference parentRegion : Region[1..1]
    // all states have "parentRegions"
    // but not all have "parentStates"
    reference parentState : State[0..1]
    // states may have a composite region
    feature region : Region[0..*]
}
target Mission {
    instance MainRegion : Region {
        // "AState" instance subsets "MainRegion.entryState"
        instance MainEntry :> entryState : State {
            // "MainEntry"'s "parentRegion" is "MainRegion"
            reference parentRegion <- MainRegion
        }
        // "AState" instance subsets "MainRegion.states"
        instance AState :> states : State {
            reference parentRegion <- MainRegion
        }
        instance BState :> states : State {
            reference parentRegion <- MainRegion
            // "BRegion" instance subsets "BState.region"
            instance BRegion :> region : Region {
                instance CState :> states : State {
                    reference parentRegion <- BRegion
                    reference parentState <- BState
                }
            }
        }
    }
}
```

Listing 4.1: The Mission target instantiates a Region with various inner states and additional inner regions.

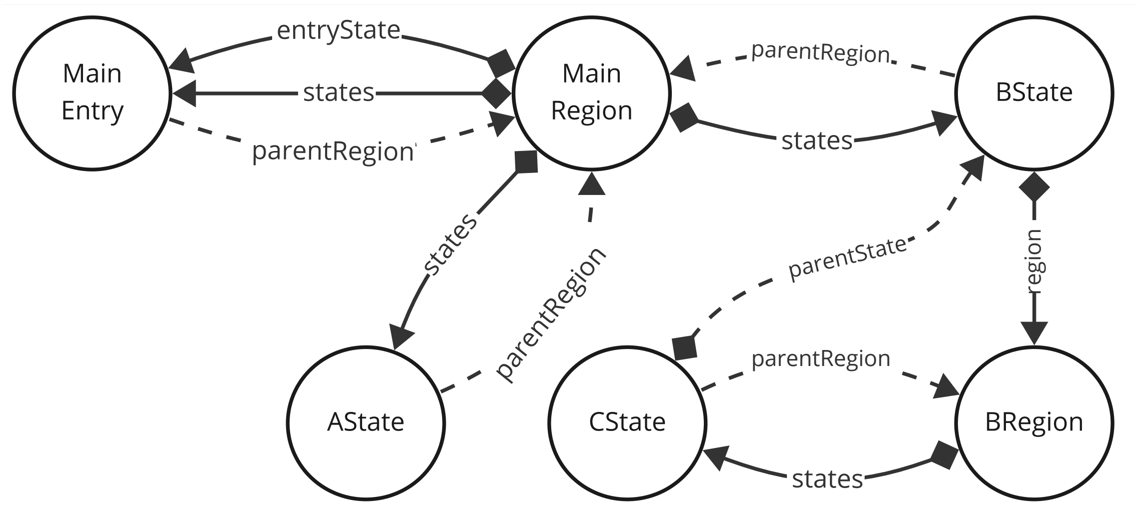

Figure 4.1: Illustration of the concrete instances and their relations in the target definition presented in Listing 4.1.

Listing 4.1 depicts a simple Region-State composition hierarchy. Regions contain States with a “states” feature. Regions also have a special “entryState” relation that subsets the “states” relation. States reference their parents using a “parentRegion” relation and a “parentState” relation. While all states must have a parent region, not all have a parent state, since in this model the containment hierarchy starts with regions. In this example, states may also contain several regions. The Mission target contains the instantiations themselves, refining the model with instance relationships subsetting the specific relations defined above. Figure 4.1 illustrates the concrete instances this model defines. Dashed lines depict references, and solid lines depict containment features. Note, that there are two relations between *MainRegion* and *MainEntry*, since *entryState* subsets the *states* feature.

4.3.6 Feature-typed Variable

It is common in existing systems to create enumerations for instances to keep track of them during execution. For example, the Region specified in Listing 4.1 has at most one active inner state at a time (zero, or one). **Feature Typed variables** are a syntax sugar achieving exactly this. Feature typings can be refactored into simple enumeration definitions containing a literal for each concrete instance the model defines. Listing 4.3.6 depicts an example of a feature-typed variable. By extending the previous example with a new *activeState* variable typed by the feature *states* with *optional* multiplicity, we can track a single state (or Nothing, if no states are active). Note, that two enumerations are created for the two concrete instances of the Region type.

```
type Region {
    ...
    feature states : State[0..*]
    var activeState : states[0..1]
}
target Mission {
    instance MainRegion : Region {
        instance AState :> states : State
        instance BState :> states : State {
            instance BRegion :> region : Region {
                instance CState :> states : State
            }
        }
    }
}
```

```
// enum from MainRegion.states
enum MainRegion_states_type {
    MainRegion_AState,
    MainRegion_BState,
    _Nothing
}
// enum from MainRegion.BState.Bregion.states
enum MainRegion_BState_BRegion_states_type {
    MainRegion_BState_BRegion_CState,
    _Nothing
}
```

Listing 4.3.6: An example of feature typed variable (left) activeState, and its corresponding actual enumeration definitions (right).

4.3.7 Static Recursion

In real-world models, recursion is frequently used, e.g., to model the state activation/deactivation of composite states. Unfortunately, XSTS does not support runtime recursion yet. However, using the already presented language features of OXSTS, one can achieve *static recursion*. Static recursion means that the recursion is implemented using repeated conditional inlinings with specific parameter bindings. In contrast to *dynamic recursion*, which would call the transitions at runtime, static recursions are unfolded at compile-time.

```
type State {
    tran enter(commonRegion: Region) {
        // if parentRegion != commonRegion specified
        inline if (parentRegion != commonRegion) {
            // enter the state above
            inline parentState.enter(commonRegion)
        }
        // set the activeState variable to "Self"
        parentRegion.activeState := Self
    }
    tran exit(commonRegion: Region) {
        // set the activeState variable to "Nothing"
        parentRegion.activeState := Nothing
        // if parentRegion != commonRegion specified
        inline if (parentRegion != commonRegion) {
            // exit the state above
            inline parentState.exit(commonRegion)
        }
    }
}
target Mission {
    tran {
        inline MainRegion.AState.enter(MainRegion)
        inline MainRegion.AState.exit(MainRegion)
        inline MainRegion.BState.BRegion.CState.enter(MainRegion)
    }
}
```

Listing 4.2: Excerpt of the OXSTS model with static recursion.

Listing 4.2 depicts an example of static recursion, implementing the composite entering/exiting of states. Before a state can be entered, all its parent states must be entered to preserve the state activation order. Thus, the enter transition contains an inline if operation initiating a static recursion, until the *commonRegion* is reached. Exit transition works likewise, only with the order flipped around: inner states must be exited before outer states. Listing 4.3 depicts the inlined version of the target's main transition.

```
target Mission {
    tran {
        // inline of MainRegion.AState.enter(MainRegion)
        MainRegion.activeState := MainRegion_AState;

        // inline of MainRegion.AState.exit(MainRegion)
        MainRegion.activeState := _Nothing;

        // inline of MainRegion.BState.BRegion.CState.enter(MainRegion)
        MainRegion.activeState := MainRegion_BState; // first the outer state,
        // then the inner state is entered
        MainRegion.BState.BRegion.activeState := MainRegion_BState_BRegion_CState;
    }
}
```

Listing 4.3: The inlined version of the static recursion example defined in Listing 4.2.

4.3.8 Composite Transition Inlining

To inline a specific transition of all instances contained in a feature, one may use *composite transition inlining*. Doing so is equivalent to inlining each instance's transition one by one in the order they are specified, except it is done automatically: there is no need to change it if a new instance is created, allowing extensions to already defined types. Composite inline operations are refactored by placing inline transition operations in the specified composite: choice in the case of *inline choice* and sequence in the case of *inline seq*. Note, that currently, it is impossible to use parameters with composite inlining.

Listing 4.4 introduces a new *Transition* type inlining the state deactivation/activation transitions of its *from* and *to* states. Statechart contains several transitions, which are all inlined in its main transition using an *inline choice* operation. Listing 4.5 contains the choice with the inline operations added.

```
type Transition {
    tran execute {
        inline from.exit(commonRegion)
        inline to.enter(commonRegion)
    }
}
type Statechart {
    feature region : Region[1..1]
    feature transitions : Transition[0..*]
    tran {
        inline choice transitions -> execute else {
            // none of the transitions can be executed
        }
    }
}
target Mission {
    instance sc : Statechart {
        instance t1 : transitions : Transition {
            ...
        }
        instance t2 : transitions : Transition ...
        instance t3 : transitions : Transition ...
        instance mainRegion :> region : Region ..
    }
    tran {
        inline sc.main()
    }
}
```

Listing 4.4: An example of composite inlining, using the composite inline choice operation.

4.3.9 Polymorphism

The word polymorphism is derived from Greek and means "having multiple forms". In OXSTS, inherited transitions may be overridden – similarly to other object-oriented programming languages. The base transition must have the *virtual* modifier in the base type, and the *override* modifier in the subtype. Overridden transitions can be overridden any amount of times, and always the last overriding transition is selected for inlining. Listing 4.6 depicts a model in which various the *AssumeFalseAction* and *AssumeTrueAction* types inherit from the *BaseAction* type, and override its main transition.

```
type Region ...
type State ...
type Transition ...
target Mission {
    instance sc : Statechart {
        instance t1 : transitions : Transition {
            ...
        }
        instance t2 : transitions : Transition ...
        instance t3 : transitions : Transition ...
        instance mainRegion :> region : Region ...
    }

    tran {
        choice {
            inline sc.t1.main()
        } or {
            inline sc.t2.main()
        } or {
            inline sc.t3.main()
        } else {
            // none of the transitions can be executed
        }
    }
}
```

Listing 4.5: The inlined version of the target definition specified in Listing 4.4.

```
type BaseAction {
    virtual tran { }
}
type AssumeFalseAction : BaseAction {
    override tran {
        assume (false)
    }
}
type AssumeTrueAction : BaseAction {
    override tran {
        assume (true)
    }
}
```

Listing 4.6: Basic transition overriding example.

Chapter 5

Semantifyr

The new Gamma→XSTS transformation workflow's (see Chapter 3) main component is Semantifyr. Semantifyr implements the OXSTS language support, and the automated mapping of OXSTS models into XSTS.

This chapter summarizes the implementation details of the Semantifyr component and the OXSTS language. The chapter is structured as follows. Section 5.1 presents the implementation philosophy of the component. The concrete transformation phases are detailed in Section 5.2.

5.1 Implementation Philosophy

In the past decades, Eclipse[1] and its plug-in ecosystem were the go-to choice for implementing similar tools. However, the industry landscape has shifted, and now there is a growing trend towards adopting new Integrated Development Environments (IDEs), such as VS Code[2] or Theia[3].

To adapt to this change, our focus was on enhancing Semantifyr's versatility across different IDEs, and thus, preventing vendor lock-in. Semantifyr was implemented from the ground up, with as minimal dependencies as possible to ensure it does not inherit the lock-in Eclipse plugins usually have.

The resulting code-base is designed to be built using the Gradle[4] build system, offering automation and efficiency. Semantifyr is implemented in Kotlin, a modern JVM-based language known for its modern features and great interoperability with other JVM-based languages, such as Java.

The XSTS artifacts that Semantifyr produces can be used in algorithms already existing in Gamma, allowing the reuse of its advanced features, such as model slicing, test-, and code generation.

[1]`https://eclipseide.org/`
[2]`https://code.visualstudio.com/`
[3]`https://theia-ide.org/`
[4]`https://gradle.org/`

5.2 Transformation Phases

Traditionally, computer program compilers exhibit five phases: lexical analysis, syntactical analysis, semantic analysis, optimization, and code generation [36]. This is no different with Semantifyr. Figure 5.1 depicts the phases of the OXSTS → XSTS transformation pipeline.

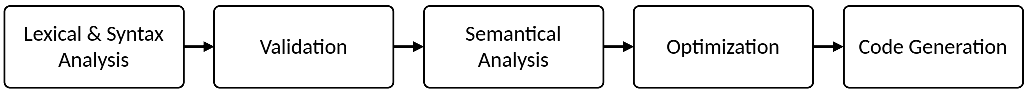

Figure 5.1: The main transformation phases in Semantifyr.

5.2.1 Lexical & Syntactical Analysis

The first step during transformation is to turn the textual input into processable models. During lexical analysis, the letters are turned into tokens, which are in turn transformed into an Abstract Syntax Tree (AST). During syntactical analysis, the AST is turned into the domain model, by connecting the cross-references across the tree.

Eclipse Modeling Framework Eclipse EMF[5] is a modeling framework and code generation toolkit for building tools and other applications based on a structured data model. Given a model specification in XMI format, EMF provides tools and runtime support to generate a set of Java classes representing the model. Additionally, it generates adapter classes that facilitate the simple editing and processing of EMF models. Importantly, EMF provides the foundation for interoperability with other EMF-based tools. The domain model of Semantifyr is specified using EMF, which enables the integration of other EMF-based tools.

Xtext Eclipse Xtext[6] is a framework for developing programming languages and domain-specific languages. It covers all aspects of a complete language infrastructure, from parsers, over linkers, to compilers. Xtext provides the means to create Eclipse-first IDEs, or general-purpose IDEs using the language server protocol (LSP). In the context of Semantifyr, an Xtext grammar is utilized to specify OXSTS. Additionally, Semantifyr uses the general-purpose LSP IDE generated by Xtext, and thus can be integrated into modern IDEs, such as VS Code and Theia.

The lexical and syntactical analysis is done by the Xtext framework.

5.2.2 Validation

The next step in the transformation is to validate the domain model. Syntactically correct models may not always adhere to the necessary semantical constraints. To ensure only semantically legal models are transformed, and to provide the users with valuable insights about incorrect models, various validation rules have been implemented.

Validation rules serve as essential checks, identifying erroneous patterns within the model. Although these patterns might be syntactically correct, they violate specific semantical

[5]https://projects.eclipse.org/projects/modeling.emf.emf
[6]https://projects.eclipse.org/projects/modeling.tmf.xtext

rules. For instance, issues such as subsetting a feature with an incompatible type, non-conformity to the multiplicity constraints of features, or assigning an integer value to a boolean variable can be detected using these rules.

Semantifyr currently implements the following simple validation rules:

- A feature's type must be compatible (equal to or inheriting from) with its subsetted feature's type.
- Feature multiplicity must be respected when defining instances in the model.
- Only `virtual` transitions may be overridden.

These validation rules are implemented using Xtext's integrated validation service and thus are provided through the general-purpose IDE.

5.2.3 Semantical Analysis

The semantical analysis phase processes the syntactically and semantically legal model, transforming all meta-programming constructs into plain XSTS constructs.

Semantical analysis is done in three steps, as depicted in Figure 5.2.

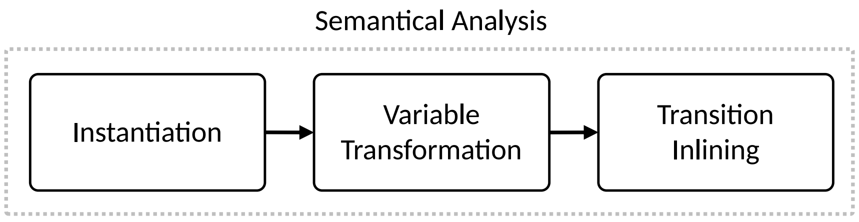

Figure 5.2: The main steps in the Semantical Analysis phase.

Instantiation First, the specific Target definition is selected, and instantiated. Instantiation means the collection of all *instance* features and the construction of concrete instances for each instance feature, as well as the instantiation of variables defined in its type. When there are no more instances to be instantiated, the reference bindings are resolved by traversing the graph. At the end of this process, a *root instance* is returned, which represents the Target definition itself. At this point, all features, references, and variables are known for each concrete instance, thus the model is considered unfolded.

Variable Transformation Feature-typed variables must be transformed into simple variables with enumeration definitions. After the instantiation process, every concrete instance is visited, and its feature-typed variables are transformed. Since all features and references are known at this point, the enumeration literals can be simply constructed by the associated concrete instances.

Transition Inlining Finally, the transitions must be inlined. At this point, the Target definition either contains only XSTS operations, in which case the transformation is finished, or contains inline operations as well. In the latter case, the inline operations are inlined into the target definition, which may result in additional inline operations – e.g., in the case of inline-choice, which results in a choice with inner inline operations. This process is repeated until there are no more inline transitions, at which point the semantical analysis is complete.

5.2.4 Optimization

After semantical analysis, the target definition only contains pure XSTS constructs. However, it may contain redundant elements – choices with a single branch, sequences inside sequences, etc. The optimization phase simplifies the model, by removing redundant.

The following simplifications are implemented currently:

- **Expression simplification**: the transformed models usually contain expressions with redundant elements, such as `or` with a `false` literal as one operand. Such expressions are simplified, by replacing the *expression* with the other operand.
- **Redundant choice-else removal**: the else branch on choices only affects the model semantics when there are branches that *may* not be able to be executed. Operations that can *always* be executed are called *always executable*. Always executable operations include assignments, havocs, and composite operations containing always executable operations, and choices with else branches. If all branches of a choice are *always executable*, then the else is redundant. This simplification is done in a bottom-up manner, meaning the innermost choices are optimized first.
- **Sequence/choice operation simplification**: composite operations – such as sequences and choices (without else branches) – with a single inner operation can be safely replaced with the inner operation itself.
- **Empty operation simplification**: sequences, choices (without else branches), if operations without any inner operations can be safely removed.
- **True assumption removal**: true assumptions do not affect the model, and thus can be safely removed.
- **Empty operation removal**: empty operations do not affect the model, and thus can be safely removed. The only exception is when they are directly inside an *else* branch, in which case it can not be removed.

Note, that only removals are done during optimizations. Equivalent model rewrites into other forms are currently not supported, since they differ from case to case which equivalent model representation performs better during model checking.

Gamma also supports extensive model reduction and model slicing techniques for XSTS, which are done later in the Gamma-XSTS transformation workflow, on the output model of Semantifyr. The optimizations in Semantifyr were designed to focus on OXSTS-related constructs primarily.

5.2.5 Code Generation

In the code generation phase, the now simplified XSTS model is serialized into its textual representation. If needed, the code generator component can output XSTS with choice-else branches rewritten into simple branches.

Choice-else rewrite An equivalent form of the else branch is one, that is only allowed to be executed, *iff* all other branches may not. To model this, specific assumptions must be placed in the branch that only evaluates to true iff all other branches cannot be executed.

This operation must also be done bottom-up, since in composite choices, the inner choice's assumptions determine the outer choice's assumptions. The assumption of an operation is defined as follows.

- Assumption of an assumption operation is the assumption itself.
- Assumption of a havoc operation is `assume (true)`.
- Assumption of an assignment operation is `assume (true)`.
- Assumption of as sequence operation is the conjugation of its inner operations' assumptions.
- Assumption of a choice operation is the disjunction of its inner operations' assumptions.

Note that we do not need to define the assumption of choice-elses, since the algorithm works bottom-up, all internal choice-elses are rewritten at this point.

During the rewrite, the negated version of the choice assumption is added to the beginning of its else branch, which assumption only evaluates to true *iff* all branches evaluate to false – since it was constructed that way. Finally, the else branch is added to the choice as a normal branch.

Note that this procedure is only possible if the assumptions do not reference variables that are set by other operations in the same branch, since that would need to be replicated in the new branch as well. Otherwise, the choice-else operations cannot be rewritten in such models.

Chapter 6

Gamma Semantic Library

To support the semantics of existing Gamma models specified with statecharts and composition (see Section 2.4) with the workflow proposed in Chapter 3, the Gamma semantics must be modeled in OXSTS. Implementing the Gamma semantic library entails the replication of the implicit model elements used in Gamma, such as *regions*, *states*, *channels*, and *components*. Using this library, the Gamma-OXSTS mapping becomes as simple as mapping all the model elements to their counterparts in the semantic library.

This chapter presents the Gamma semantic library in Section 6.1, and showcases its usage through an example model in Section 6.2.

6.1 Semantic Library

The most basic type is the *Event* type. Events are represented with a single boolean flag, as shown in Listing 6.1. The Event type exposes various transitions for easy manipulation, e.g., the *isSet* transition with the *isActive* assumption.

```
type Event {
    var isActive: Boolean := false
    tran set {
        isActive := true
    }
    tran reset {
        isActive := false
    }
    havoc {
        havoc (isActive)
    }
    tran isSet {
        assume (isActive)
    }
}
```

Listing 6.1: The Event type in the Gamma Semantic Library.

```
type Timeout {
    var deltaTime : Integer := -1
    tran {
        if (deltaTime >= 0) {
            deltaTime := deltaTime - 1
        }
    }
    tran isUp {
        assume (deltaTime = 0)
    }
}
```

Listing 6.2: The Timeout type in the Gamma Semantic Library.

Timeouts (modeled in Listing 6.2) are modeled using an integer variable that is counting down to -1. To prime the timeout, one has to set the *deltaTime* variable to any value. The *main* transition decreases the variable by one until it reaches -1. The timeout is considered *up* when the timer variable equals zero. Note that modeling it this way ensures the timeout only fires once, exactly at 0.

Events and Timeouts are primarily used by Triggers. Triggers (see Listing 6.3), such as EventTriggers and TimeoutTriggers, refer to events and timeouts respectively. Their *isTriggered* transition can be used as guards, only allowing execution when the event or transition the trigger references has occurred or is up.

```
type Trigger {
    virtual tran isTriggered { }
}
type EventTrigger : Trigger {
    reference event : Event[1..1]
    override tran isTriggered {
        inline event.isSet()
    }
}
type TimeoutTrigger : Trigger {
    reference timeout : Timeout[1..1]
    override tran isTriggered {
        inline timeout.isUp()
    }
}
```

Listing 6.3: The Trigger type in the Gamma Semantic Library.

```
type Action {
    virtual tran { }
}
type RaiseEventAction : Action {
    reference event : Event[1..1]
    override tran {
        inline event.set()
    }
}
type SetTimeoutAction : Action {
    reference timeout : Timeout[1..1]
}
```

Listing 6.4: The Action type in the Gamma Semantic Library.

For the manipulation of events and timeouts during the execution of the model, Actions may be used. Listing 6.4 defines the *Action* type and its sub-types: *RaiseEventAction* and *SetTimeoutAction*. Note, that at the time of writing, it is impossible to forward values or variable references to instances, thus separate *SetTimeoutAction* refinement types are needed for various values.

Components represent the central elements of the composition language in Gamma (see Listing 6.5). *CompositeComponents* may contain other components and channels. *Channels* are modeled as a simple event forwarding type, proxying the *inputEvent* to the *outputEvent*. Synchronous components are modeled by the *SyncComponent* type. SyncComponents are initialized by sequentially initializing inner components, and their main transition executes the inner transitions one by one, and then runs each of the channels. This semantic is the same as defined in Section 2.4.

Statecharts are special *Components*. Defined in Listing 6.6, Statecharts contain events, timeouts, and regions. Events are grouped in two categories, *inputEvents* and *outputEvents*. The statechart component first resets all its outputEvents, to clear them before execution. Next, all region transitions are fired. Since at this point the inputEvents have been used, they can be cleared. Finally, the timeouts are executed, meaning their deltaTime variables are decreased.

```
type Component {
    virtual init { }
    virtual tran { }
}
type Channel {
    reference inputEvent : Event[1..1]
    reference outputEvent : Event[1..1]
    tran {
        if (inputEvent.isActive) {
            inline outputEvent.set()
        }
    }
}
type CompositeComponent : Component {
    feature components : Component[0..*]
    feature channels : Channel[0..*]
}
type SyncComponent : CompositeComponent {
    override init {
        inline seq components -> init
    }
    override tran {
        inline seq components -> main
        inline seq channels -> main
    }
}
```

Listing 6.5: The Component type hierarchy in the Gamma Semantic Library.

```
type Statechart : Component {
    feature events : Event[0..*]
    feature inputEvents :> events : Event
     [0..*]
    feature outputEvents :> events : Event
     [0..*]
    feature timeouts : Timeout[0..*]
    feature regions : Region[0..*]
    override init {
        inline seq regions ->
     activateRecursive
    }
    override tran {
        inline seq outputEvents -> reset
        inline choice regions ->
            fireTransitions else { }
        inline seq inputEvents -> reset
        inline seq timeouts -> main
    }
}
```

Listing 6.6: The Statechart type in the Gamma Semantic Library.

Next, the *Region* type is defined in Listing 6.7. Regions contain states, entry- and simple transitions. Regions also have a feature-typed variable, *activeState*, typed by the *states* feature. Using feature typing, all Regions automatically have an enumeration variable specifying their active state (or Nothing, when deactivated).

Note the workaround at *fireTransitionsInner*. At this point, the language does not allow the fine-grained control required for the exact modeling of general Gamma Regions. For this reason, a workaround transition was needed, that uses an *inline choice* without an *else* branch. This is important since the else branch means the transitions do not need to be executed every time, which is not in accordance with the Gamma statechart semantics.

States implement most of the state-transition behavior of Gamma statecharts. As shown in Listing 6.8, the *State* type references the parent region and the parent state. States also may contain several regions and entry-exit actions. Exiting and entering are implemented using *static recursion*. First, the exit actions are executed, and then the containing region's activeState variable is set to *Nothing*. Afterwards, the static recursion is initiated with an inline if, checking whether the containing region is the commonRegion specified. The Common region is the lowest region that is a parent to both the source and the target of the transition. The *deactivateRecursive* transition is called when an upper state is left, since in such cases the lower states must be exited as well.

```
type Region {
    feature states : State[0..*]
    feature abstractTransitions : AbstractTransition[0..*]
    feature transitions :> abstractTransitions : Transition[0..*]
    feature entryTransitions :> abstractTransitions : EntryTransition[0..*]
    var activeState : states[0..1] := Nothing
    tran activateRecursive {
        inline seq entryTransitions -> main
    }
    tran deactivateRecursive {
        inline seq states -> deactivateRecursive
    }
    tran fireTransitions {
        inline choice transitions -> main else {
            inline choice states -> fireTransitions
        }
    }
    tran fireTransitionsInner { // workaround
        inline choice transitions -> main
    }
}
```

Listing 6.7: The Region type in the Gamma Semantic Library.

```
type State {
    reference parent : Region[1..1]
    reference parentState : State[0..1]
    feature regions : Region[0..*]
    feature entryActions : Action[0..*]
    feature exitActions : Action[0..*]
    tran isActive {
        assume (parent.activeState = Self)
    }
    tran deactivateRecursive {
        inline seq regions -> deactivateRecursive
        if (parent.activeState = Self) {
            parent.activeState := Nothing
            inline seq exitActions -> main
        }
    }
    tran exitRecursive(commonRegion : Region) {
        inline seq exitActions -> main
        parent.activeState := Nothing
        inline if (commonRegion != parent) {
            inline parentState.exitRecursive(commonRegion)
        }
    }
    tran exit(commonRegion : Region) {
        inline exitRecursive(commonRegion)
        inline seq regions -> deactivateRecursive
    }
    tran enterRecursive(commonRegion : Region) {
        inline if (commonRegion != parent) {
            inline parentState.enterRecursive(commonRegion)
        }
        parent.activeState := Self
        inline seq entryActions -> main
    }
    tran enter(commonRegion : Region) {
        inline enterRecursive(commonRegion)
        inline seq regions -> activateRecursive
    }
    tran fireTransitions {
        inline seq regions -> fireTransitionsInner
    }
}
```

Listing 6.8: The State type in the Gamma Semantic Library.

```
type AbstractTransition {
    reference commonRegion : Region[1..1]
    reference to : State[1..1]
    virtual tran { }
}
type EntryTransition : AbstractTransition {
    override tran {
        inline to.enter(commonRegion)
    }
}
```

Listing 6.9: Abstract and entry transitions in the Gamma Semantic Library.

```
type Transition : AbstractTransition {
    reference from : State[1..1]
    feature trigger : Trigger[1..1]
    feature actions : Action[0..*]
    tran {
        inline trigger.isTriggered()
        inline from.isActive()
        inline from.exit(commonRegion)
        inline seq actions -> main
        inline to.enter(commonRegion)
    }
}
```

Listing 6.10: The Transition type in the Gamma Semantic Library.

Transitions, as defined in Listing 6.9 refine the *AbstractTransition* type. The *EntryTransition* is a special transition that does not have a source state.

Finally, the *Transition* type is defined in Listing 6.10. Transitions contain a single trigger and several associated actions. Transitions check that their trigger has been triggered, and their source state is active. If the checks succeed, the source state is *exited*, and the target state is *entered*. The *actions* are executed in between to keep conformance with the Gamma semantics.

6.2 Example Model

Constructing a statechart with Gamma semantics in OXSTS is done by refining the *Statechart* and *Component* types, and instantiating them in a custom *Target* definition.

This example models the synchronous composite system defined in Listing 2.3 using the Gamma Semantic Library.

Listing 6.11 depicts the OXSTS version of the Leader statechart defined in Listing 2.1. The *fire*, *stop*, and *start* events are modeled separately and are placed in the *inputEvents* and *outputEvents* features depending upon their direction. The regions and states are modeled using Type instantiations subsetting the relevant features. Note, that the Gamma semantic library does not define an entry state, so the entry transitions only specify a *to* state.

Similarly to the Leader, Listing 6.13 depicts the OXSTS version of the Worker statechart defined in Listing 2.2.

Listing 6.12 models the composition of the two statecharts, by refining the *SyncComponent* type.

Finally, Listing 6.14 defines the verification environment of the System component. First, the instantiated system is initialized. The main transition simulates the environment by inlining the *havoc* operation of the two input ports of the Leader statechart. Afterwards, the System is executed. The Mission's invariant property specifies that the Worker.Operational state is never active. The Theta model checker (see Section 2.3.2) can read to XSTS version of this model, and either prove the invariant is always true or show a counter-example in the form of an execution trace. In this sense, checking whether a given state is reachable can be checked by stating the state is never active.

```
type LeaderStatechart : Statechart {
    instance fireEvent :> inputEvents : Event
    instance stopEvent :> inputEvents : Event
    instance startEvent :> outputEvents : Event
    instance Main :> regions : Region {
       instance et1 :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Idle
        }
        instance Idle :> states : State {
            reference parent <- Main
        }
        instance idleToOperational :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Idle
            reference to <- Operational
            instance t1Trigger :> trigger : EventTrigger {
                reference event <- fireEvent
            }
            instance t1Action :> actions : RaiseEventAction {
                reference event <- startEvent
            }
        }
        instance operationalToIdle :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Operational
            reference to <- Idle
            instance t2Trigger :> trigger : EventTrigger {
                reference event <- stopEvent
            }
        }
        instance Operational :> states : State {
            reference parent <- Main
        }
    }
}
```

Listing 6.11: The OXSTS version of the Leader statechart defined in Listing 2.1.

```
type System : SyncComponent {
    instance leader :> components : LeaderStatechart
    instance worker :> components : WorkerStatechart
    instance startChannel :> channels : Channel {
        reference inputEvent <- leader.startEvent
        reference outputEvent <- worker.startEvent
    }
}
```

Listing 6.12: The OXSTS version of the synchronous System defined in Listing 2.3.

```
type WorkerStatechart : Statechart {
    instance startEvent :> inputEvents : Event
    instance Main :> regions : Region {
        instance et :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Idle
        }
        instance Idle :> states : State {
            reference parent <- Main
        }
        instance idleToOperational :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Idle
            reference to <- Operational
            instance t1Trigger :> trigger : EventTrigger {
                reference event <- startEvent
            }
        }
        instance Operational :> states : State {
            reference parent <- Main
        }
    }
}
```

Listing 6.13: The OXSTS version of the Worker statechart defined in Listing 2.2.

```
target Mission {
    instance system : System
    init {
        inline system.init()
    }
    tran {
        // simulating the environment
        havoc (system.leader.fireEvent.isActive)
        havoc (system.leader.stopEvent.isActive)
        // executing the system
        inline system.main()
    }
    prop {
        // invariant, stating worker.operational is never active
        system.worker.Main.activeState !=
            system.worker.Main.Operational
    }
}
```

Listing 6.14: The target definition specifying the mission environment of Listing 6.12.

Chapter 7

Evaluation

Since Semantifyr and OXSTS are intended to be used in the critical systems domain, the evaluation of the approach and implementation on example models is a must.

The chapter is structured as follows. In Section 7.1, an automatic conformance test suite is defined, checking the correct implementation of Semantifyr. Section 7.2 introduces the Crossroads System, and evaluates the OXSTS language experimentally. Lastly, Section 7.3 summarizes the evaluation results.

7.1 Conformance Tests

To validate the implementation, I constructed an automated conformance test suite. The primary goal of the test suite is to provide a complete coverage of all OXSTS meta constructs. By specifying input and expected OXSTS-XSTS pairs, Semantifyr's conformance to the OXSTS specification can be validated. The test suite (Table 7.1) lists 44 handcrafted models categorized by the language feature they test. Each test is a standalone model focusing on a specific OXSTS feature.

All implemented conformance tests successfully validated the functionality of the Semantifyr transformation component. These tests not only confirmed Semantifyr's correct operation but also helped during development, automatically indicating broken features.

7.2 Gamma Semantic Library - Case Study

To validate the OXSTS language and the Gamma semantic library, a case study is performed on the Gamma semantic library. The Crossroads [10] example model is used as an example from the Gamma GitHub repository.

7.2.1 Crossroads System

The Crossroads system (see Figure 7.1) models an imagined vehicle crossroad traffic light system. The crossroad is made up of two pairs of traffic lights, each controlled by a central Controller component. Each traffic light uses the standard 3-phase light system looping through the Red → Green → Yellow → Red sequence. Additionally, there is an *interrupted* mode that may be triggered by the police. When interrupted, the lights are blinking in Yellow: Blank → Yellow → Blank.

Category	Name
Feature	Feature Subseting
	Reference Subseting
Feature Typing	Multiple Variables
	Multiple Variables - Different Feature
	Single Variable
	Variable with Nothing
	Variable with Nothing expression
Inheritance	Base Type Feature
	Child Type Feature
	Multiple Overrides
	Multiple Types
	Non-virtual transformation is not overridden
	Virtual transformation is overridden
Inline Transition	Inline call of multiple transitions
	Inline call with no instance
	Inline call with parameters
	Inline choice with no instances
	Inline composite choice
	Inline composite sequence
	Inline functor
	Inline if false
	Inline if true
	Inline seq with no instances
	Inline static recursion
	Mulitple inline calls
	Single inline call
Instance	Flat target with multiple instantiations
	Flat target with single instantiation
	Many layer composite instantiation
	Many layer composite reference binding
	Mulitple instances in one reference
	Reference binding
	Reference binding to upper
	Two-layer composite instantiation
	Two-layer composite reference binding
Target	Boolean Variable
	Enum Variable
	Integer Variable
Transition	Assignment
	Choice
	Choice else
	Havoc
	If
	sequence

Table 7.1: Automated conformance tests categorized by the tested language feature.

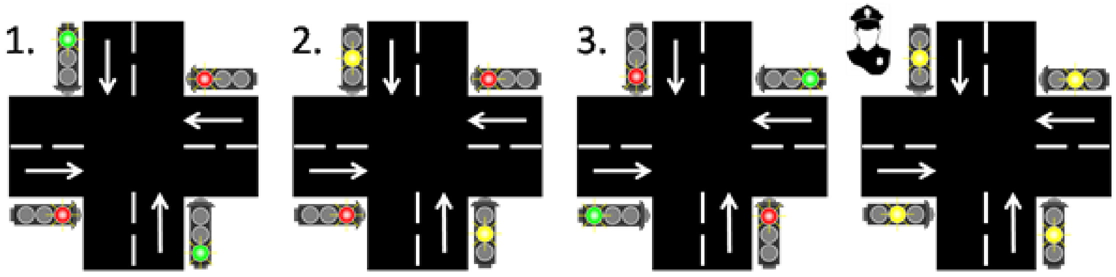

Figure 7.1: The behavior of the Crossroads System from the Gamma Tutorial model [10].

The system is controlled by a central Crossroads Controller component, as depicted in Figure 7.2. The Crossroads Controller component is made up of a *Controller* component and two *TrafficLightController* components. During normal operation, the Controller component sends *toggle* events to the *TrafficLightController* components through the *Control* ports. Upon each toggle, the TrafficLights switch to the next light in the sequence, by sending a specific value through the *Lights* port. The Controller component forwards all incoming *Police* events to each TrafficLightController and stops sending toggle events until the next police event.

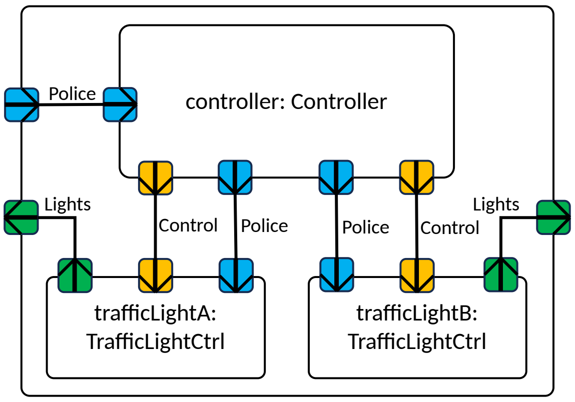

Figure 7.2: The Crossroads Controller component.

The SysML representation of the TrafficLigthCtrl component is shown in Figure 7.3. There are two main states: Normal and Interrupted. The operation starts in state Normal, and switches to Interrupted upon a *Police* event. The Normal state is composite, meaning it has inner states. The inner states model the cycle of the lights: Red, then Green, then Yellow, then start again. The interrupted state contains two inner states: Blank and BlinkingYellow. The state machine switches between these two states every second. Note, that the transitions between Interrupted and Normal have a higher priority than inner transitions.

The SysML model of the Controller component is shown in Figure 7.4. There are two main states: Operating and Interrupted. The operation starts in state Operating, and switches to Interrupted upon a *Police* event. In the Operating state, the first state is Init. After one second, the component switches to TrafficOnA state and sends a *toggle* event to TrafficLightA (switching it to Yellow). Next, after two seconds, the component switches to the StoppingA state and sends yet another toggle event to TrafficLightA (switching it to Green). This cycle continues through TrafficOnB and StoppingB and finally returns to TrafficOnA. Upon a Police event, the execution switches to the Interrupted state, which

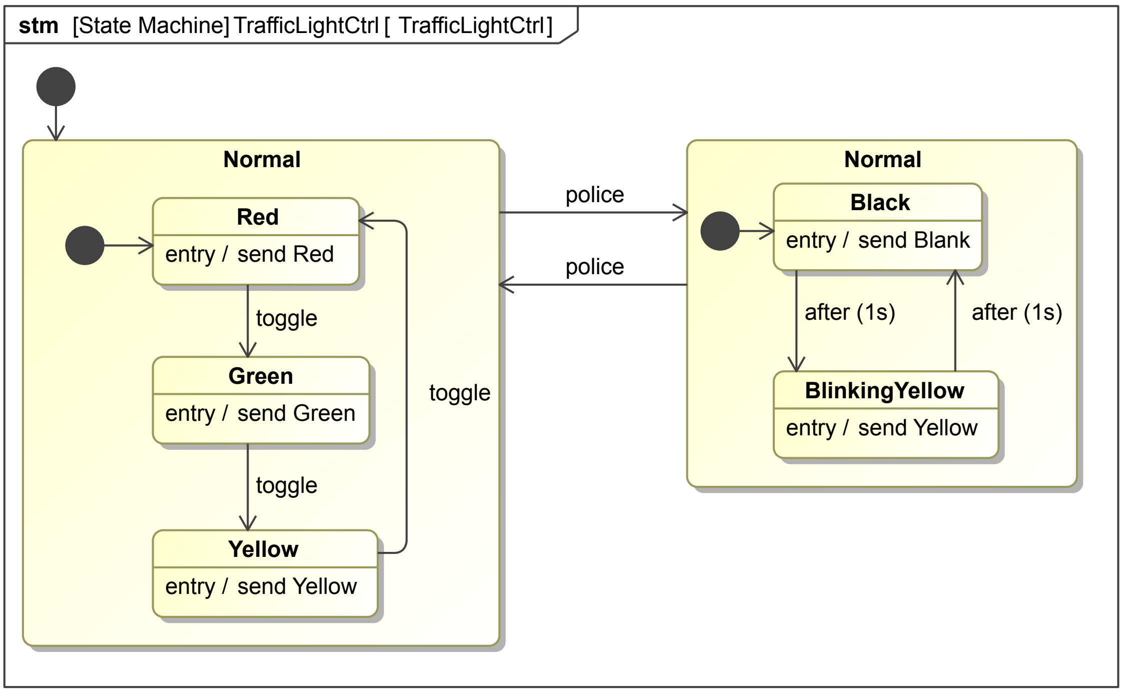

Figure 7.3: The TrafficLightCtrl State Machine model.

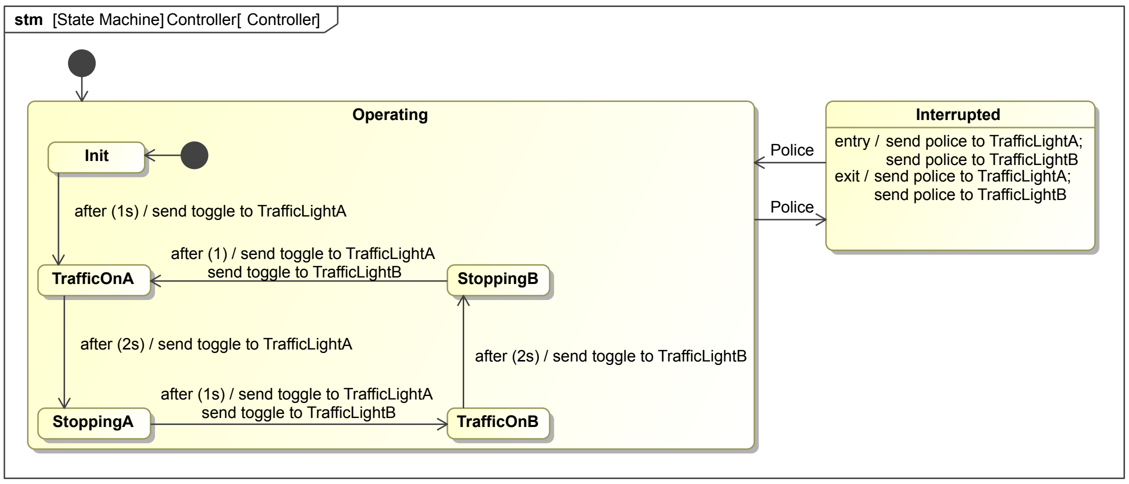
stm [State Machine] Controller[Controller]
Operating
Init
after (1s) / send toggle to TrafficLightA
TrafficOnA
after (1) / send toggle to TrafficLightA
send toggle to TrafficLightB
StoppingB
after (2s) / send toggle to TrafficLightA
after (2s) / send toggle to TrafficLightB
StoppingA
after (1s) / send toggle to TrafficLightA
send toggle to TrafficLightB
TrafficOnB
Police
Police
Interrupted
entry / send police to TrafficLightA;
send police to TrafficLightB
exit / send police to TrafficLightA;
send police to TrafficLightB

Figure 7.4: The Controller State Machine model.

forwards the police event to the two TrafficLight components. Note that the transition between Operating and Interrupted has a higher priority than internal ones.

7.2.2 Gamma Model

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

Listing 7.1: The interface definitions of the Crossroads system [27].

To evaluate the OXSTS model, a reference model is needed in Gamma representation. The Gamma models were taken from the Gamma GitHub repository [10] and were used with slight modifications. for more information on Gamma, see Section 2.4. First, the interfaces are defined in Listing 7.1. Gamma groups events together in reusable interfaces, which are later used to define ports of components.

Listing 7.2 shows the Gamma model of the Controller component. Note the TrafficTimeout, which is used to model the *elapsed time* semantics of the original model.

Similarly to the previous, Listing 7.3 shows the Gamma model of the TrafficLightCtrl component.

The final Crossroad Controller component is defined in Listing 7.4. Note the use of synchronous composition semantics in this model, which is not an exact replication of the original behavior. Previous work defined the exact execution semantics of SysML state machines by modeling it in Gamma [33]. However, this will be sufficient for this work as an initial evaluation of the semantic library.

7.2.3 OXSTS Model

Using the Gamma semantic library modeled in Section 6.1, the Crossroad model can be replicated in OXSTS.

An excerpt of the Controller component model is shown in Listing 7.5. The various events are instantiated directly in the type, and placed into the input-output events feature, depending upon their direction.

The Operating state is continued in Listing 7.6. Note, that the triggers and the actions are all modeled by *instantiating* them inside the states and transitions.

The traffic light controller is modeled similarly. An excerpt of the model is shown in Listing 7.7.

The complete source of the models can be viewed in Section A.1, however, the presented excerpts are sufficient for understanding the rest of the work.

```
statechart Controller [
    port Police : requires PoliceInterrupt
    port ControlA : provides Control
    port PoliceA : provides PoliceInterrupt
    port ControlB : provides Control
    port PoliceB : provides PoliceInterrupt
] {
    timeout TrafficTimeout // timeout specification
    region Main {
        initial Entry0
        state Operating {
            region operating {
                initial Entry1
                state Init {
                    // entry action of Init state
                    entry / set TrafficTimeout := 1 s;
                }
                state TrafficOnA {
                    entry / set TrafficTimeout := 2 s;
                }
                state StoppingA {
                    entry / set TrafficTimeout := 1 s;
                }
                state TrafficOnB {
                    entry / set TrafficTimeout := 2 s;
                }
                state StoppingB {
                    entry / set TrafficTimeout := 1 s;
                }
            }
        }
        state Interrupted {
            entry / raise PoliceA.police; raise PoliceB.police;
            exit / raise PoliceA.police; raise PoliceB.police;
        }
    }

    transition from Entry0 to Operating
    // ...
    transition from StoppingB to TrafficOnA when timeout TrafficTimeout
        / raise ControlA.toggle; raise ControlB.toggle;
}
```

Listing 7.2: The Controller component behavior modeled in the Gamma language [27].

```
statechart TrafficLightCtrl [
    port Control : requires Control
    port Police : requires PoliceInterrupt
    port Light : provides LightCommands
] {
    timeout BlinkingTimeout
    region Main {
        initial Entry0
        state Normal {
            region normal {
                initial Entry1
                state Red {
                    entry / raise Light.displayRed;
                }
                state Green {
                    entry / raise Light.displayGreen;
                }
                state Yellow {
                    entry / raise Light.displayYellow;
                }
            }
        }
        state Interrupted {
            region interrupted {
                initial Entry2
                state Black {
                    entry / set BlinkingTimeout := 1 s;
                        raise Light.displayNone;
                }
                state BlinkingYellow {
                    entry / set BlinkingTimeout := 1 s;
                        raise Light.displayYellow;
                }
            }
        }
    }
    transition from Entry0 to Normal
    transition from Normal to Interrupted when PoliceInterrupt.police
    // ...
    transition from Black to BlinkingYellow when timeout BlinkingTimeout
}
```

Listing 7.3: The TrafficLightCtrl component behavior modeled in Gamma [27].

```
sync CrossroadController [
    port police : requires PoliceInterrupt,
    port lightA : provides LightCommands,
    port lightB : provides LightCommands
] {
    // declaring internal components
    component controller : Controller
    component trafficLightA : TrafficLightCtrl
    component trafficLightB : TrafficLightCtrl
    // binding input and output ports to internal components
    bind police -> controller.Police
    bind lightA -> trafficLightA.Light
    bind lightB -> trafficLightB.Light
    // Connecting ports of components using channels
    channel [controller.ControllA] -o)- [trafficLightA.Control]
    channel [controller.ControllB] -o)- [trafficLightB.Control]
    channel [controller.ControllA] -o)- [trafficLightB.PoliceInterrupt]
    channel [controller.ControllB] -o)- [trafficLightB.PoliceInterrupt]
}
```

Listing 7.4: The Crossroad Controller component modeled in Gamma [27].

```
type Controller : Statechart {
    instance policeEvent :> inputEvents : Event
    instance policeEventA :> outputEvents : Event
    instance toggleEventA :> outputEvents : Event
    instance policeEventB :> outputEvents : Event
    instance toggleEventB :> outputEvents : Event
    instance trafficTimeout :> timeouts : Timeout
    instance Main :> regions : Region {
        instance et1 :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Operating
        }
        instance Operating :> states : State {
            reference parent <- Main
            // ...
        }
        instance operatingToInterrupted :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Operating
            reference to <- Interrupted
            instance t :> trigger : EventTrigger {
                reference event <- policeEvent
            }
        }
        instance interruptedToOperating :> transitions : Transition // ...
        instance Interrupted :> states : State {
            reference parent <- Main
            instance ea1 :> entryActions : RaiseEventAction {
                reference event <- policeEventA
            }
            instance ea2 :> entryActions : RaiseEventAction {
                reference event <- policeEventB
            }
            instance ea3 :> exitActions : RaiseEventAction {
                reference event <- policeEventA
            }
            instance ea4 :> exitActions : RaiseEventAction {
                reference event <- policeEventB
            }
        }
    }
}
```

Listing 7.5: Excerpt of the Controller behaviour modeled in OXSTS using the Gamma Semantic library.

```
instance Operating :> states : State {
    reference parent <- Main
    instance OperatingRegion :> regions : Region {
        instance et2 :> entryTransitions : EntryTransition {
            reference commonRegion <- OperatingRegion
            reference to <- Init
        }
        instance Init :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating
            instance ea1 :> entryActions : SetTimeoutActionOneS {
                reference timeout <- trafficTimeout
            }
        }
        instance initToTrafficOnA :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Init
            reference to <- TrafficOnA
            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a :> actions : RaiseEventAction {
                reference event <- toggleEventA // switch A to Green
            }
        }
        instance TrafficOnA :> states : State // ...
        instance trafficOnAToStoppingA :> transitions : Transition // ...
        instance StoppingA :> states : State // ...
        instance stoppingAToTrafficOnB :> transitions : Transition // ...
        instance TrafficOnB :> states : State // ...
        instance trafficOnBToStoppingB :> transitions : Transition // ...
        instance StoppingB :> states : State // ...
        instance stoppingBToTrafficOnA :> transitions : Transition // ...
    }
}
```

Listing 7.6: Excerpt of the Operating state of the Controller statechart modeled in OXSTS.

```
type TrafficLightCtrl : Statechart {
    instance policeEvent :> inputEvents : Event
    instance toggleEvent :> inputEvents : Event
    instance displayRedEvent :> outputEvents : Event
    instance displayYellowEvent :> outputEvents : Event
    instance displayGreenEvent :> outputEvents : Event
    instance displayNoneEvent :> outputEvents : Event
    instance blinkingTimeout :> timeouts : Timeout
    instance Main :> regions : Region {
        instance et :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Normal
        }
        instance Normal :> states : State {
            reference parent <- Main
            // ...
        }
        instance normalToInterrupt :> transitions : Transition // ...
        instance interruptToNormal :> transitions : Transition // ...
        instance Interrupted :> states : State {
            reference parent <- Main
        }
    }
}
```

Listing 7.7: Excerpt of the TrafficLightCtrl behaviour modeled in OXSTS using the Gamma Semantic library.

7.2.4 Experimental Evaluation

To validate the conformance of the OXSTS model behavior to the Gamma behavior, I specified several formal requirements. Assuming both models are black boxes, if the user can not differentiate between them by their behavior, then the models can be considered conforming. The behavior is tested by defining several formal requirements, and testing them using the Theta model checker (see Section 2.3.2).

Semantic Conformance

The most substantial part of the semantic library is the state-transition part, which defines how states, regions, and transitions behave. To validate this part of the semantic library, I defined formal properties asserting that the semantic library respects the semantics of composite states.

- If a composite state is active, all composite states and their internal states must be inactive.
- If a composite state is active, one, and only one internal state must be active.

The following requirements are formalized for the Crossroad System.

- If Controller.Interrupted is active, then the OperatingRegion must not have any active states;
- If Controller.Operating is active, then exactly one of Init, TrafficOnA, StoppingA, TrafficOnB, or StoppingB must be active;
- If TrafficLightCtrl.Normal is active, then Interrupted.InterruptedRegion must not have any active states;
- If TrafficLightCtrl.Interrupted is active, then Normal.NormalRegion must not have any active states;
- If TrafficLightCtrl.Normal is active, then exactly one of Red, Green, or Yellow must be active;
- If TrafficLightCtrl.Interrupted is active, then exactly one of Black or BlinkingYellow must be active;

The TrafficLightCtrl requirements must be duplicated for the two instances: TrafficLightA and TrafficLightB. The Target definitions can be found at Section A.2.

Behavioral Conformance

The behavioral conformance of the models can be validated with requirements based on the expected behavior of the system. The following requirements are formalized from the description of the Crossroads System.

- All states must be reachable;
- Both TrafficLightCtrls must be Interrupted at the same time;

- Only allowed pairs of lights must be active, see Table 7.2.

Table 7.2 lists all state-configurations the two TrafficLightCtrl components may be in during Normal operation, annotated with whether it is allowed or not. Since the two lights follow a strict sequence of states, some state configurations are impossible in a faithful representation.

TrafficLightA	**TrafficLightB**	**Is Allowed**
Red	Red	✓
Red	Green	✓
Red	Yellow	✓
Green	Red	✓
Green	Green	✗
Green	Yellow	✗
Yellow	Red	✓
Yellow	Green	✗
Yellow	Yellow	✗

Table 7.2: All state-configurations the two TrafficLightCtrl components may be in (during Normal operation) annotated with if allowed.

The Target definitions with the behavioral conformance properties can be found at Section A.2.

Conformance Results

After modeling the semantic and behavioral conformance requirements, a total of 40 formal properties are constructed. I executed the Theta model checker on all properties, using the Gamma model as a baseline. The constructed target definitions can be viewed in Section A.2, postfixed with either safe or unsafe, depending on the expected result. From the 40 requirements, all 40 produced the expected results, both for the Gamma model and the OXSTS model.

Error Detection

Using the above-defined conformance results, we can detect errors in the model. To test the detection capabilities, a variant of the Controller component was made. Listing 7.8 presents the changes in the model. By removing some RaiseEventActions, the police event is no longer forwarded to the TrafficLightB component, thus the two Trafficlights can get out of sync with each other.

After re-running the model checker, the following errors can be found.

- TrafficLightB.Interrupted is not reachable;
- TrafficLightB.Interrupted.Black is not reachable;
- TrafficLightB.Interrupted.Yellow is not reachable;
- Both TrafficLightCtrls cannot be interrupted at the same time;
- TrafficLightA is Green and TrafficLightB is Green at the same time;

```
instance Interrupted :> states : State {
    reference parent <- Main
    instance ea1 :> entryActions : RaiseEventAction {
        reference event <- policeEventA
    }
    // Removed lines:
//    instance ea2 :> entryActions : RaiseEventAction {
//        reference event <- policeEventB
//    }
    instance ea3 :> exitActions : RaiseEventAction {
        reference event <- policeEventA
    }
    // Removed lines:
//    instance ea4 :> exitActions : RaiseEventAction {
//        reference event <- policeEventB
//    }
}
```

Listing 7.8: Removed lines in the Interrupted state of the Controller component.

- TrafficLightA is Green and TrafficLightB is Yellow at the same time;
- TrafficLightA is Yellow and TrafficLightB is Green at the same time;
- TrafficLightA is Yellow and TrafficLightB is Yellow at the same time;

7.3 Conclusion

This section encapsulates the evaluation of OXSTS through the Gamma semantic library case study presented in Section 7.2, highlighting critical aspects and addressing areas of improvement.

Crossroads System evaluation The successful modeling of the moderately complex Crossroads System example model marks a significant milestone. The Crossroads System contains various difficult-to-model features, such as composite states, timeouts, various entry-exit and transition actions, composition, and channel semantics. The comprehensive evaluation campaign validated the Gamma semantic library. Although more work and more complex examples are needed, this example provides assurance, that the proposed approach has viability in the real world.

OXSTS vs Gamma languages With experience gained from previous works developing new languages for the Gamma framework, and the modeling of the Gamma semantic library in OXSTS, I can compare the two.

With Gamma, one has to understand the plugin and code architecture to pinpoint the exact location where the code must be extended/changed. Of course, this is the same with OXSTS, since a theoretical Semantic Library would need to be extended with additional types and refinements to introduce new languages. However, with OXSTS, the engineer can mentally stay in the same domain: transition systems.

Along this note, Gamma has various implicit semantic types, such as Components and Actions, whose behavior is difficult to understand at first glance. This is the great benefit of using OXSTS: however complicated the language semantics are, everything is defined explicitly in the same language, and can be viewed directly.

On the other side, however, Gamma is capable of representing fairly complex behavior, like asynchronous composition, and phase-statecharts, which will be difficult to replicate in OXSTS.

Note that by using Semantifyr, the complexity of Gamma is traded for complexity in Semantifyr. Further research is needed to assess the exact complexity required in the OXSTS language to replicate all of the Gamma semantics.

Missing language features Due to Semantifyr being in the prototype phase, several features essential for comprehensive modeling are absent in the current version of OXSTS. Notable among these is the lack of integration with query languages, forcing the user to manually set various references, that otherwise could be calculated automatically. Examples could include the `parent` and `parentState` references in the example model, which could be set using simple model queries, using Viatra [2] for example.

Additionally, OXSTS does not fully support the utilization of values and variables within meta-features, hindering the language's flexibility. E.g., the *SetTimeoutAction* cannot be used as is, it must be refined using subtypes. Using *Lambda* types could ease the modeling efforts since many of the used custom Action types only define a single transition.

Furthermore, advanced constructs, such as inline-for or inline-while loops, which are essential for custom transition structures, are also missing. Another interesting feature could be feature-typed variable invoking. Since feature-typed variables represent concrete instances in the OXSTS model, one could be able to use it to refer to the concrete instance, e.g., inline one of its transitions. Doing so would result in a choice operation checking which concrete instance is referred, to choose the correct transition based on the concrete instance's type. This feature would allow more dynamic behavior support.

KerML and SysML v2 support feature overrides, which allows for more intricate modeling of complex systems. Using feature overrides, one could define types, in which it is not yet defined whether a feature is a composition or a reference. The feature then could be later refined in subtypes.

Finally, due to the prototype nature of Semantifyr, several advanced XSTS constructs are missing from OXSTS, e.g., records, for and while loops (not inline), and local variables. To support the full set of the Gamma functionalities, these advanced constructs must be implemented in OXSTS as well.

Requirement satisfaction The following list evaluates the satisfaction of the requirements specified in Section 3.3.

1. **Formal Semantics**: the formal semantics of XSTS were successfully kept, and Theta was able to process it successfully.
2. **Familiarity to Engineers**: by using well-known language features, such as types, inheritance, transition overriding, and features, the language is simple to understand. However, since the language semantics are still based upon XSTS, the learning curve is steep in the beginning. Also, the language is very verbose, thus models can be very long. More user research is needed to evaluate this point.
3. **Modularity**: The language is without a doubt highly modular. Using inheritance, composition, and transition overriding, the example model successfully demonstrated the language's modularity.

4. **Expressiveness**: As mentioned previously, the language lacks several language features to be generally useable. However, the language seems to be expressive enough to model moderately complex example models.

In summary, OXSTS and Semantifyr show potential at realizing the proposed transformation workflow in Chapter 3. However, further refinement and development are needed, including the incorporation of essential features and further example models, to realize its full capability as a robust meta-programming language with configurable semantics.

Chapter 8

Conclusion and Future Work

The precise execution semantics of modeling languages are essential in the world of MBSE. For hidden formal methods to gain widespread use, advanced, and configurable formal verification tools are needed. This work addresses the challenges faced by the Gamma Statechart Composition Framework, a verification tool designed for engineering models. To simplify the framework's complexity, a new language called Objective XSTS (OXSTS) is introduced, integrating meta-programming techniques. OXSTS extends the eXtended Symbolic Transition System (XSTS) language used in Gamma with features such as types, inheritance, composition, and transition inlining. Using OXSTS, engineers can organize components into reusable libraries, streamlining the mapping of engineering languages into a simple one-to-one mapping. The use of such reusable semantic libraries allows the easy customization and extension of language semantics, optimization of formal models, and simple introduction of new languages into the Gamma framework.

The results of the work are the following. Using the new Semantifyr component, the model transformation workflow of Gamma can be extended to use OXSTS, and corresponding semantical libraries to verify high-level languages. As an evaluation, the Crossroads example system has been modeled in the OXSTS formalism. The semantic and behavioral conformance of the OXSTS model has been validated using several formal requirements. This evaluation demonstrated the applicability and the value of the approach. Using Semantifyr, 1) the customization of language semantics can be achieved by modifying the underlying semantical library; 2) the formal model can be experimentally optimized since the user has direct access to the semantic library; and 3) new language support can be achieved by the implementation of a new semantic library and a mapper component.

Future work The use of Semantifyr and OXSTS seems to be a promising way to mitigate the complexity of the Gamma framework. As a direct next step, I intend to implement additional features into the language. Integration with query languages, such as the Viatra Query Language would automate the setting of various references. Support for values and variables in meta-features would enable additional flexibility in handling dynamic data assignments. Additionally, more advanced programming constructs are needed, such as inline-for and inline-while loops for custom inline structures, lambda types for reducing the boiler-plate code, feature-typed variable invoking, allowing enhanced dynamic behavior support, and finally, feature overrides, the later refinement of features. The language is also missing several advanced XSTS constructs, such as records, for and while loops, and local variables. Finally, real-world industrial example models are crucial for a comprehensive evaluation.

Chapter 9

Acknowledgement

I would like to express my acknowledgment to my advisors, Vince Molnár and Bence Graics, who have continuously provided me with guidance, valuable ideas, and feedback during this work. Their help is invaluable and much appreciated!

Supported by the **ÚNKP-23-2-III-BME-47** New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Kulturális és Innovációs Minisztérium

National Research, Development and Innovation Office Hungary

Új Nemzeti Kiválóság Program

Bibliography

[1] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL https://doi.org/10.1007/978-3-319-10575-8_11.

[2] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, pages 101–110, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21155-8.

[3] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

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

[5] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, London, Cambridge, 1999. ISBN 0-262-03270-8.

[6] Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Templatable metamodels for semantic variation points. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Model Driven Architecture- Foundations and Applications*, pages 68–82, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-72901-3.

[7] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

[8] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017. DOI: 10.1145/3110278. URL https://doi.org/10.1145/3110278.

[9] Márton Elekes, Vince Molnár, and Zoltán Micskei. Assessing the specification of modelling language semantics: a study on UML PSSM. *Software Quality Journal*, 31 (2):575–617, Jun 2023. ISSN 1573-1367. DOI: 10.1007/s11219-023-09617-5. URL https://doi.org/10.1007/s11219-023-09617-5.

[10] Gamma. Crossroads model. https://github.com/ftsrg/gamma/tree/master/tutorial, 2019. Accessed: 2023-11-02.

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

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

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

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

[15] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable cegar framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 158–174, Cham, 2016. Springer International Publishing. ISBN 978-3-319-39570-8.

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

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

[18] Thomas A. Henzinger and Rupak Majumdar. A classification of symbolic transition systems. In Horst Reichel and Sophie Tison, editors, *STACS 2000*, pages 13–34, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-46541-6.

[19] Benedek Horváth, Vince Molnár, Bence Graics, Ákos Hajdu, István Ráth, Ákos Horváth, Robert Karban, Gelys Trancho, and Zoltán Micskei. Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering*, n/a(n/a), 2023. DOI: https://doi.org/10.1002/sys.21679. URL https://incose.onlinelibrary.wiley.com/doi/abs/10.1002/sys.21679.

[20] INCOSE. INCOSE Systems Engineering. https://www.incose.org/systems-engineering, 2023. Accessed: 2023-11-02.

[21] INCOSE. INCOSE Systems Engineering Vision 2035. https://www.incose.org/docs/default-source/se-vision/incose-se-vision-2035-executive-summary.pdf, 2023. Accessed: 2023-11-02.

[22] Saul A. Kripke. Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963. DOI: https://doi.org/10.1002/malq.19630090502. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.19630090502.

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

[24] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Comput. Surv.*, 52(6), oct 2019. ISSN 0360-0300. DOI: 10.1145/3354584. URL https://doi.org/10.1145/3354584.

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

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

[27] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: Design, verification and code generation for component-based reactive systems. In *Proceedings of ICSE'18: Companion Proceeedings*, pages 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

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

[29] OMG. *Kernel Modeling Language (KerML)*, 2023. ptc/23-06-01.

[30] OMG. *OMG System Modeling Language (SysML v2)*, 2023. ptc/23-06-02.

[31] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, sep 1977. ISSN 0360-0300. DOI: 10.1145/356698.356702. URL https://doi.org/10.1145/356698.356702.

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

[33] Péter Szkupien and Ármin Zavada. Formal Methods for Better Standards: Validating the UML PSSM Standard about State Machine Semantics. Thesis for students' scientific conference, BME, 2022.

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

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

[36] Reinhard Wilhelm, Dieter Maurer, et al. *Compiler design.* Springer, 1995.

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

Appendix A

OXSTS Models

In this chapter, a comprehensive collection of OXSTS artifacts developed during this work is presented. While not essential for understanding the core concepts of the work, delving into these artifacts can provide additional insights into the OXSTS language.

The chapter is structured as follows. In Section A.1, the OXSTS representation of the full Crossroads model is presented, introduced in Section 7.2, while Section A.2 presents the target definitions used during the evaluation of the language (see Section 7.2.4).

A.1 Crossroads Model Implementation

```
type SetTimeoutActionOneS : SetTimeoutAction {
    override tran {
        timeout.deltaTime := 1
    }
}
type SetTimeoutActionTwoS : SetTimeoutAction {
    override tran {
        timeout.deltaTime := 2
    }
}
```

Listing A.1: Custom timeout actions used in the model.

```
type Controller : Statechart {
    instance policeEvent :> inputEvents : Event

    instance policeEventA :> outputEvents : Event
    instance toggleEventA :> outputEvents : Event

    instance policeEventB :> outputEvents : Event
    instance toggleEventB :> outputEvents : Event

    instance trafficTimeout :> timeouts : Timeout

    instance Main :> regions : Region {
        instance et1 :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Operating
        }

        instance Operating :> states : State {
            reference parent <- Main

            // ...
        }

        instance operatingToInterrupted :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Operating
            reference to <- Interrupted

            instance t :> trigger : EventTrigger {
                reference event <- policeEvent
            }
        }

        instance interruptedToOperating :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Interrupted
            reference to <- Operating

            instance t :> trigger : EventTrigger {
                reference event <- policeEvent
            }
        }

        instance Interrupted :> states : State {
            reference parent <- Main

            instance ea1 :> entryActions : RaiseEventAction {
                reference event <- policeEventA
            }
            instance ea2 :> entryActions : RaiseEventAction {
                reference event <- policeEventB
            }
            instance ea3 :> exitActions : RaiseEventAction {
                reference event <- policeEventA
            }
            instance ea4 :> exitActions : RaiseEventAction {
                reference event <- policeEventB
            }
        }
    }
}
```

Listing A.2: The Controller model implementation.

```
instance Operating :> states : State {
    reference parent <- Main

    instance OperatingRegion :> regions : Region {
        instance et2 :> entryTransitions : EntryTransition {
            reference commonRegion <- OperatingRegion
            reference to <- Init
        }

        instance Init :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating

            instance ea1 :> entryActions : SetTimeoutActionOneS {
                reference timeout <- trafficTimeout
            }
        }

        instance initToTrafficOnA :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Init
            reference to <- TrafficOnA

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a :> actions : RaiseEventAction {
                reference event <- toggleEventA // switch A to Green
            }
        }

        instance TrafficOnA :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating

            instance a :> entryActions : SetTimeoutActionTwoS {
                reference timeout <- trafficTimeout
            }
        }

        instance trafficOnAToStoppingA :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- TrafficOnA
            reference to <- StoppingA

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a :> actions : RaiseEventAction {
                reference event <- toggleEventA // switch A to Yellow
            }
        }

        instance StoppingA :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating

            instance a :> entryActions : SetTimeoutActionOneS {
                reference timeout <- trafficTimeout
            }
        }
    }
}
```

Listing A.3: The first part of the Operating state definition in the Contoller component.

```
instance Operating :> states : State {
    reference parent <- Main

    instance OperatingRegion :> regions : Region {
        instance stoppingAToTrafficOnB :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- StoppingA
            reference to <- TrafficOnB

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a1 :> actions : RaiseEventAction {
                reference event <- toggleEventA // switch A to Red
            }
            instance a2 :> actions : RaiseEventAction {
                reference event <- toggleEventB // switch B to Green
            }
        }
        instance TrafficOnB :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating

            instance a :> entryActions : SetTimeoutActionTwoS {
                reference timeout <- trafficTimeout
            }
        }
        instance trafficOnBToStoppingB :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- TrafficOnB
            reference to <- StoppingB

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a :> actions : RaiseEventAction {
                reference event <- toggleEventB // switch B to Yellow
            }
        }
        instance StoppingB :> states : State {
            reference parent <- OperatingRegion
            reference parentState <- Operating

            instance a :> entryActions : SetTimeoutActionOneS {
                reference timeout <- trafficTimeout
            }
        }
        instance stoppingBToTrafficOnA :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- StoppingB
            reference to <- TrafficOnA

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- trafficTimeout
            }
            instance a1 :> actions : RaiseEventAction {
                reference event <- toggleEventB // switch B to Red
            }
            instance a2 :> actions : RaiseEventAction {
                reference event <- toggleEventA // switch A to Green
            }
        }
    }
}
```

Listing A.4: The second part of the Operating state definition in the Contoller component.

```
type TrafficLightCtrl : Statechart {
    instance policeEvent :> inputEvents : Event
    instance toggleEvent :> inputEvents : Event

    instance displayRedEvent :> outputEvents : Event
    instance displayYellowEvent :> outputEvents : Event
    instance displayGreenEvent :> outputEvents : Event
    instance displayNoneEvent :> outputEvents : Event

    instance blinkingTimeout :> timeouts : Timeout

    instance Main :> regions : Region {
        instance et :> entryTransitions : EntryTransition {
            reference commonRegion <- Main
            reference to <- Normal
        }

        instance Normal :> states : State {
            reference parent <- Main

            // ...
        }

        instance normalToInterrupt :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Normal
            reference to <- Interrupted

            instance t :> trigger : EventTrigger {
                reference event <- policeEvent
            }
        }

        instance interruptToNormal :> transitions : Transition {
            reference commonRegion <- Main
            reference from <- Interrupted
            reference to <- Normal

            instance t :> trigger : EventTrigger {
                reference event <- policeEvent
            }
        }

        instance Interrupted :> states : State {
            reference parent <- Main

            // ...
        }
    }
}
```

Listing A.5: The TrafficLightCtrl model implementation.

```
instance Normal :> states : State {
    reference parent <- Main

    instance NormalRegion :> regions : Region {
        instance et2 :> entryTransitions : EntryTransition {
            reference commonRegion <- NormalRegion
            reference to <- Red
        }

        instance Red :> states : State {
            reference parent <- NormalRegion
            reference parentState <- Normal

            instance e :> entryActions : RaiseEventAction {
                reference event <- displayRedEvent
            }
        }

        instance redToGreen :> transitions : Transition {
            reference commonRegion <- NormalRegion
            reference from <- Red
            reference to <- Green

            instance t :> trigger : EventTrigger {
                reference event <- toggleEvent
            }
        }

        instance Green :> states : State {
            reference parent <- NormalRegion
            reference parentState <- Normal

            instance e :> entryActions : RaiseEventAction {
                reference event <- displayGreenEvent
            }
        }

        instance greenToYellow :> transitions : Transition {
            reference commonRegion <- NormalRegion
            reference from <- Green
            reference to <- Yellow

            instance t :> trigger : EventTrigger {
                reference event <- toggleEvent
            }
        }

        instance Yellow :> states : State {
            reference parent <- NormalRegion
            reference parentState <- Normal

            instance e :> entryActions : RaiseEventAction {
                reference event <- displayYellowEvent
            }
        }

        instance yellowToRed :> transitions : Transition {
            reference commonRegion <- NormalRegion
            reference from <- Yellow
            reference to <- Red

            instance t :> trigger : EventTrigger {
                reference event <- toggleEvent
            }
        }
    }
}
```

Listing A.6: The Normal state definition in the TrafficLightCtrl component.

```
instance Interrupted :> states : State {
    reference parent <- Main

    instance InterruptedRegion :> regions : Region {
        instance et :> entryTransitions : EntryTransition {
            reference commonRegion <- InterruptedRegion
            reference to <- Black
        }

        instance Black :> states : State {
            reference parent <- InterruptedRegion
            reference parentState <- Interrupted

            instance ea1 :> entryActions : SetTimeoutActionOneS {
                reference timeout <- blinkingTimeout
            }
            instance ea2 :> entryActions : RaiseEventAction {
                reference event <- displayNoneEvent
            }
        }

        instance blackToYellow :> transitions : Transition {
            reference commonRegion <- InterruptedRegion
            reference from <- Black
            reference to <- Yellow

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- blinkingTimeout
            }
        }

        instance yellowToBlack :> transitions : Transition {
            reference commonRegion <- InterruptedRegion
            reference from <- Yellow
            reference to <- Black

            instance t :> trigger : TimeoutTrigger {
                reference timeout <- blinkingTimeout
            }
        }

        instance Yellow :> states : State {
            reference parent <- InterruptedRegion
            reference parentState <- Interrupted

            instance ea1 :> entryActions : SetTimeoutActionOneS {
                reference timeout <- blinkingTimeout
            }
            instance ea2 :> entryActions : RaiseEventAction {
                reference event <- displayYellowEvent
            }
        }

    }
}
```

Listing A.7: The Interrupted state definition in the TrafficLightCtrl component.

```
type CrossroadController : SyncComponent {
    instance controller :> components : Controller
    instance trafficLightA :> components : TrafficLightCtrl
    instance trafficLightB :> components : TrafficLightCtrl

    instance policeAChannel :> channels : Channel {
        reference inputEvent <- controller.policeEventA
        reference outputEvent <- trafficLightA.policeEvent
    }
    instance toggleAChannel :> channels : Channel {
        reference inputEvent <- controller.toggleEventA
        reference outputEvent <- trafficLightA.toggleEvent
    }

    instance policeBChannel :> channels : Channel {
        reference inputEvent <- controller.policeEventB
        reference outputEvent <- trafficLightB.policeEvent
    }
    instance toggleBChannel :> channels : Channel {
        reference inputEvent <- controller.toggleEventB
        reference outputEvent <- trafficLightB.toggleEvent
    }
}
```

Listing A.8: The implementation of the CrossroadController component

A.2 Crossroads Target Definitions

```
abstract target CrossroadsMission {
    instance crossroad : CrossroadController

    init {
        inline crossroad.init()
    }

    tran {
        inline crossroad.controller.policeEvent.havoc
        inline crossroad.main()
    }
}
```

Listing A.9: The base target definition for the Crossroad system.

```
target ControllerOperatingInterruptedExclusive_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.controller.Main.activeState =
                crossroad.controller.Main.Interrupted &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                Nothing
        )
    }
}
target ControllerOperatingIncorrectStateHierarchy_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.controller.Main.activeState = crossroad.controller.Main.Operating &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                crossroad.controller.Main.Operating.OperatingRegion.Init &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                crossroad.controller.Main.Operating.OperatingRegion.TrafficOnA &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                crossroad.controller.Main.Operating.OperatingRegion.StoppingA &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                crossroad.controller.Main.Operating.OperatingRegion.TrafficOnB &&
            crossroad.controller.Main.Operating.OperatingRegion.activeState !=
                crossroad.controller.Main.Operating.OperatingRegion.StoppingB
        )
    }
}
```

Listing A.10: The semantic requirements of the Crossroads component.

```
target TrafficLightANormalInterruptedExclusive_Safe : CrossroadsMission {
    prop {
        ! (
            (
                crossroad.trafficLightA.Main.activeState =
                    crossroad.trafficLightA.Main.Normal &&
                crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.activeState !=
                    Nothing
            ) || (
                crossroad.trafficLightA.Main.activeState =
                    crossroad.trafficLightA.Main.Interrupted &&
                crossroad.trafficLightA.Main.Normal.NormalRegion.activeState !=
                    Nothing
            )
        )
    }
}
target TrafficLightANormalIncorrectStateHierarchy_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.activeState =
                crossroad.trafficLightA.Main.Normal &&
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightA.Main.Normal.NormalRegion.Red &&
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightA.Main.Normal.NormalRegion.Green &&
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightA.Main.Normal.NormalRegion.Yellow
        )
    }
}
target TrafficLightAInterruptedIncorrectStateHierarchy_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.activeState =
                crossroad.trafficLightA.Main.Interrupted &&
            crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.activeState !=
                crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.Black &&
            crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.activeState !=
                crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.Yellow
        )
    }
}
```

Listing A.11: The semantic requirements of the TrafficLightA component.

```
target TrafficLightBNormalInterruptedExclusive_Safe : CrossroadsMission {
    prop {
        ! (
            (
                crossroad.trafficLightB.Main.activeState =
                    crossroad.trafficLightB.Main.Normal &&
                crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.activeState !=
                    Nothing
            ) || (
                crossroad.trafficLightB.Main.activeState =
                    crossroad.trafficLightB.Main.Interrupted &&
                crossroad.trafficLightB.Main.Normal.NormalRegion.activeState !=
                    Nothing
            )
        )
    }
}
target TrafficLightBNormalIncorrectStateHierarchy_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightB.Main.activeState =
                crossroad.trafficLightB.Main.Normal &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightB.Main.Normal.NormalRegion.Red &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightB.Main.Normal.NormalRegion.Green &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState !=
                crossroad.trafficLightB.Main.Normal.NormalRegion.Yellow
        )
    }
}
target TrafficLightBInterruptedIncorrectStateHierarchy_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightB.Main.activeState =
                crossroad.trafficLightB.Main.Interrupted &&
            crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.activeState !=
                crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.Black &&
            crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.activeState !=
                crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.Yellow
        )
    }
}
```

Listing A.12: The semantic requirements of the TrafficLightB component.

```
target ControllerOperating_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.activeState =
            crossroad.controller.Main.Operating)
    }
}
target ControllerOperatingInit_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.Operating.OperatingRegion.activeState =
            crossroad.controller.Main.Operating.OperatingRegion.Init)
    }
}
target ControllerOperatingTrafficOnA_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.Operating.OperatingRegion.activeState =
            crossroad.controller.Main.Operating.OperatingRegion.TrafficOnA)
    }
}
target ControllerOperatingStoppingA_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.Operating.OperatingRegion.activeState =
            crossroad.controller.Main.Operating.OperatingRegion.StoppingA)
    }
}
target ControllerOperatingTrafficOnB_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.Operating.OperatingRegion.activeState =
            crossroad.controller.Main.Operating.OperatingRegion.TrafficOnB)
    }
}
target ControllerOperatingStoppingB_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.Operating.OperatingRegion.activeState =
            crossroad.controller.Main.Operating.OperatingRegion.StoppingB)
    }
}
target ControllerInterrupted_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.controller.Main.activeState =
            crossroad.controller.Main.Interrupted)
    }
}
```

Listing A.13: Target definitions checking that each state of Controller is reachable.

```
target TrafficLightANormal_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.activeState =
            crossroad.trafficLightA.Main.Normal)
    }
}
target TrafficLightANormalRed_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightA.Main.Normal.NormalRegion.Red)
    }
}
target TrafficLightANormalGreen_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightA.Main.Normal.NormalRegion.Green)
    }
}
target TrafficLightANormalYellow_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightA.Main.Normal.NormalRegion.Yellow)
    }
}
target TrafficLightAInterrupted_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.activeState =
            crossroad.trafficLightA.Main.Interrupted)
    }
}
target TrafficLightAInterruptedBlack_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.activeState =
            crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.Black)
    }
}
target TrafficLightAInterruptedYellow_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.activeState =
                crossroad.trafficLightA.Main.Interrupted.InterruptedRegion.Yellow
        )
    }
}
```

Listing A.14: Target definitions checking that each state of TrafficLightA is reachable.

```
target TrafficLightBNormal_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.activeState =
            crossroad.trafficLightB.Main.Normal)
    }
}
target TrafficLightBNormalRed_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightB.Main.Normal.NormalRegion.Red)
    }
}
target TrafficLightBNormalGreen_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightB.Main.Normal.NormalRegion.Green)
    }
}
target TrafficLightBNormalYellow_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
            crossroad.trafficLightB.Main.Normal.NormalRegion.Yellow)
    }
}
target TrafficLightBInterrupted_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.activeState =
            crossroad.trafficLightB.Main.Interrupted)
    }
}
target TrafficLightBInterruptedBlack_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.activeState =
            crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.Black)
    }
}
target TrafficLightBInterruptedYellow_Unsafe : CrossroadsMission {
    prop {
        ! (crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.activeState =
            crossroad.trafficLightB.Main.Interrupted.InterruptedRegion.Yellow)
    }
}
```

Listing A.15: Target definitions checking that each state of TrafficLightB is reachable.

```
target ARedBRed_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Red &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Red
        )
    }
}
target ARedBGreen_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Red &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Green
        )
    }
}
target ARedBYellow_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Red &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Yellow
        )
    }
}
```

Listing A.16: The requirements specified in Table 7.2. Part 1.

```
target AGreenBRed_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Green &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Red
        )
    }
}
target AGreenBGreen_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Green &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Green
        )
    }
}
target AGreenBYellow_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Green &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Yellow
        )
    }
}
```

Listing A.17: The requirements specified in Table 7.2. Part 2.

```
target AYellowBRed_Unsafe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Yellow &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Red
        )
    }
}
target AYellowBGreen_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Yellow &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Green
        )
    }
}
target AYellowBYellow_Safe : CrossroadsMission {
    prop {
        ! (
            crossroad.trafficLightA.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightA.Main.Normal.NormalRegion.Yellow &&
            crossroad.trafficLightB.Main.Normal.NormalRegion.activeState =
                crossroad.trafficLightB.Main.Normal.NormalRegion.Yellow
        )
    }
}
```

Listing A.18: The requirements specified in Table 7.2. Part 3.

```
target BothMustBeInterruptedAtTheSameTime_Unsafe : CrossroadsMission {
    prop {
        ! (
            (
                crossroad.trafficLightA.Main.activeState =
                    crossroad.trafficLightA.Main.Interrupted &&
                crossroad.trafficLightB.Main.activeState =
                    crossroad.trafficLightB.Main.Interrupted
            )
        )
    }
}
target BothMustBeInterruptedAtTheSameTime_Safe : CrossroadsMission {
    prop {
        ! (
            (
                crossroad.trafficLightA.Main.activeState =
                    crossroad.trafficLightA.Main.Interrupted &&
                crossroad.trafficLightB.Main.activeState !=
                    crossroad.trafficLightB.Main.Interrupted
            ) || (
                crossroad.trafficLightA.Main.activeState !=
                    crossroad.trafficLightA.Main.Interrupted &&
                crossroad.trafficLightB.Main.activeState =
                    crossroad.trafficLightB.Main.Interrupted
            )
        )
    }
}
```

Listing A.19: The behavioral requirements of the Crossroad System.