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

A Modular Modelling Language for Configurable Formalisation and Verification of Engineering Models

Master's Thesis

Author
Ármin Zavada

Advisor
dr. Vince Molnár

June 3, 2024

Contents

Kivonat

A modellalapú rendszertervezés területén egyre nagyobb igény mutatkozik a formális módszerek alkalmazására. Ez azonban olyan szakértelmet igényel, amely a rendszermérnököktől nem várható el. Ennek eredményeképpen szükség van magas szintű mérnöki nyelvekről alacsony szintű formalizmusokba történő automatizált leképezésekre. Bár történtek kísérletek e szakadék áthidalására, kérdések még mindig fennállnak. Az ontológiai nyelvek irányába történő elmozdulással a rendszereket 4D-s előfordulási osztályokként modellezik, nem pedig egy idővel fejlődő 3D-s rendszerként, ami akadályozza a korszerű modellellenőrző algoritmusok alkalmazását. Az ilyen ontológiai viselkedésmodellek ontológiai következtetéseket igényelnek, amelyek nem tudják hatékonyan kezelni az állapottér-robbanást. Az ilyen nyelvek operacionalizálása segítene, de az ontológiai nyelvek operacionalizálásakor az eredeti szemantika konformitását még az optimalizációk jelenlétében is validálnunk kell. Mindennek tetejébe ezeket a kihívásokat minden egyes új mérnöki nyelv, változat vagy variáns esetében meg kell oldani.

A fenti problémák javítására egy alternatív megközelítést javaslok, amelyet a Kernel Modeling Language inspirált. E megközelítés célja, hogy a magas szintű mérnöki modellek szemantikáját egy új ontológiai-operacionális modellezési nyelv bevezetésével visszavezessük az alacsony szintű elemek szemantikájára. Ez a nyelv egy alacsony szintű formális nyelvből származik, és egy metaprogramozási konstrukciók által vezérelt kompozíciós transzformációs motorral van felszerelve, ami megkönnyíti a magas szintű nyelvek szemantikus modellezését. A magas szintű mérnöki nyelvek szemantikájának modellezésével szemantikus könyvtárak készíthetők. A magas szintű nyelvek leképezése így modellezési, és nem modellátalakítási feladattá válik, ahol a magas szintű nyelv szemantikus variánsai az újrafelhasználható szemantikus könyvtárakból a megfelelő modellelemek kiválasztásával finomíthatók. Az ilyen szemantikus könyvtárak ontológiai nyelvekbe történő közvetlen integrálásával az ontológiai szemantika automatikus operacionalizálása érhető el.

Ebben a munkában bemutatom az OXSTS-t és a Semantifyr-t, egy új ontológiai-operációs modellező nyelvet és a hozzá tartozó eszközt, amelyek képesek ilyen szemantikus könyvtárak modellezésére és levetítésére egy elemzési formalizmusba. Bemutatok továbbá egy új megközelítést az ontológiai nyelvek verifikálására szemantikus könyvtárak alkalmazásával. Megközelítésem demonstrálására lemodellezem a Gamma szemantikáját egy szemantikus könyvtárban, majd annak konformitását megvizsgálom kettő magas szintű rendszeren.

Abstract

In the field of model-based systems engineering, there is an increasing demand for the application of formal methods. However, this requires expertise which cannot be expected from systems engineers. As a result, there is a need for automated transformation from high-level engineering languages to low-level formalisms. While attempts have been made to bridge this gap, questions still remain. With the trend shifting towards ontological languages, systems are modeled as classes of 4D occurrences, rather than a 3D system evolving with time, which hinders the application of state-of-the-art model checking algorithms. Such ontological behavior models need ontological reasoning, which cannot handle the state space explosion efficiently. Operationalizing such languages would help, but when operationalizing ontological languages, we need to validate the conformance of the two semantics, even in the presence of optimizations. On top of all, these challenges must also be solved for every new engineering language, version, or variant.

To mitigate the above issues, I propose an alternative approach inspired by the Kernel Modeling Language. This approach aims to trace the semantics of the high-level engineering models back to the semantics of low-level elements through the introduction of a new ontological-operational modeling language. Derived from a low-level formal language and equipped with a compositional transformation engine driven by meta-programming constructs, this language facilitates the modeling of high-level language semantics. By modeling the semantics of high-level engineering languages, semantic libraries can be constructed, which allow the direct modification of the language semantics. 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 the reusable semantic libraries. By integrating such semantic libraries into ontological languages directly, an automatic operationalization of the ontological semantics is achieved.

In this work, I present OXSTS and Semantifyr, a new ontology-operational modeling language and an associated tool capable of modeling such semantic libraries and mapping them back to an analysis formalism. I also present a new approach to verify ontological languages using semantic libraries. To demonstrate my approach, I model the semantics of Gamma in a semantic library and then investigate its conformance on two high-level systems.

Chapter 1

Introduction

The application of formal methods in model-based systems engineering (MBSE) is becoming increasingly prevalent [21], as the introduction of models in the design process provides a straightforward opportunity to use automatic reasoning techniques on the knowledge base describing the system. The expected benefit is a better understanding and, often, the formal verification of the system's properties throughout the design process, starting from the requirement engineering phase and extending to the implementation and even the operation phases (e.g., diagnostics [56]).

A well-known challenge in formal verification is using formal languages to describe the design in a sufficiently precise form. Using such languages and the tools processing them generally requires a high level of expertise in formal methods, which is not expected from systems engineers [41]. There have been several attempts to bridge this gap by establishing mappings from high-level engineering languages to formal languages [58, 37, 43, 12]. While these approaches definitely helped in increasing the adoption of formal methods in MBSE, there are many open questions and repeating challenges that prevent its widespread application [52, 36, 25].

One of the theoretical challenges comes from the fact that engineering languages are mostly ontological in nature. They focus on the structure of models, with declarative semantics based on classification. While behavior models traditionally had operational semantics (even in UML [28] and SysML v1 [27], except Sequence Diagrams [42]), the latest trends display a shift towards ontological behavior modeling [7]. Version 2 of the Systems Modeling Language (SysML v2) [47] and the underlying Kernel Modeling Language (KerML) [46] have fully declarative semantics formalized with a set of axioms.

Ontological behavior modeling treats models as classes of 4D occurrences rather than a 3D system evolving with time [7]. This kind of semantics makes it easy to check if a given trace (i.e., a complete execution of a system) conforms to the model but does not give guidance on how to compute such an execution. On the contrary, operational semantics specify an algorithm to compute a temporal unfolding of the system behavior, which makes it easy to execute a model but does not provide an easy way to check the conformance of an execution and the model [44]. While there are ontology reasoners that can deal with declarative semantics, the efficiency of verification tools requiring operational semantics is generally vastly superior, partly due to decades of research improving their performance.

Another typical challenge for formal verification is the state space explosion problem [55, 14], i.e., even small models may produce a state space too large to represent in memory. To address this phenomenon, verification algorithms usually try to exploit the high-level structure of models (e.g., symmetries or common patterns) to compress the state space or

optimize its computation [1]. Another typical (partial) solution is to limit the engineering language to a subset that can be efficiently operationalized, excluding model elements that could inflate the state space (e.g., arbitrary concurrency).

A recurring question that is hard to answer is how to prove that the mapping from the high-level engineering language to the formal analysis language preserves the semantics. This is even harder in the presence of domain-specific optimizations, which often result in more abstract execution traces – thus the state-space reduction.

On top of all the challenges mentioned above, an extra layer of technical difficulties comes from the fact that these challenges have to be solved with every new engineering language, a new version or variant, or even for custom semantics used by companies (mostly because of an existing tool integration, such as a simulator).

In this work, I use SysML v2 [47] and KerML [46] to examine these challenges and formulate a vision about a way forward, ultimately leading to efficient verification of ontology-based modeling languages. My proposed solution is based on encoding the operationalized semantics in the ontological knowledge base. This way, we can still use ontological derivations to reduce high-level models to a suitable abstraction level, where instead of further reduction and unfolding of the temporal aspects, we can map into the concepts of a formal language with operational semantics instead. This level can vary with the target formal language and the capabilities of the corresponding verification engine. There are several opportunities to cross-check the semantic conformance of the different derivation and verification paths. By encoding the operationalization in the ontology, we reduce the transpilation aspect between the two languages to a simple syntactical translation, while the transformation of the semantics is left to the ontology reasoner, with the necessary reduction rules encoded in semantic libraries. These libraries become part of the knowledge base, resulting in flexibility when defining or customizing the operational semantics for a given model. Additionally, using such semantic libraries is not constrained to ontological languages; the semantics of classic operational languages (e.g., SysML) can be modeled as well, leading to a simpler way of supporting such languages and language variants.

My main contributions are the following.

- I propose a workflow of using semantic libraries to operationalize ontological languages.
- I introduce a new ontological-operational language called OXSTS, and the Semantifyr tool capable of interpreting it.
- I model a subset of the Gamma behavioral semantics in a semantic library.
- I conduct a preliminary evaluation of the approach and the modeled semantic library using high-level models.
- Finally, I draw my conclusions about the approach and outline the way forward.

The rest of the work is structured as follows. Chapter 2 gives an overview of the theoretical background needed to understand the main contributions. Chapter 3 gives an overview of the new approach. Next, Chapter 4 details the Semantifyr implementation, including the formulation of the OXSTS language. In Chapter 5 an initial Gamma Semantic Library is implemented, mapping the implicit model elements in Gamma to OXSTS types. Next, Chapter 6 evaluates the Gamma semantic library through case studies. Finally, Chapter 7 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. First, Section 2.1 introduces the concept of ontologies. Next, Section 2.2 overviews MBSE languages, and introduces the SysML v2 and KerML languages. In Section 2.3, the theoretical background of model checking is presented. Section 2.4 introduces the XSTS language, a formal analysis language that can be used during model checking. Lastly, Section 2.5 introduces the Gamma Statechart Composition Framework, which enables the formal modeling and verification of component-based reactive systems.

2.1 Ontology

In philosophy, Ontology is the branch of metaphysics that studies the nature of being, existence, and reality [30]. It addresses fundamental questions about what entities exist, how they can be categorized, and the relationships between them. In information technology, ontology (with lowercase "o") refers to a structured framework for representing knowledge within a specific domain, involving the explicit specification of concepts, classes, relationships, and rules. Usually, ontologies define a specific *classification* over a set of *things* in the *universe* of the ontology.

Classification is absolute, however, what various actors of the universe care about are highly subjective. Figure 2.1 depicts an example of three people, Alice, Bob and Charlie. The people are classified as *human*, however, Alice and Bob are classified as *woman* and *man* respectively, while Charlie is classified as a *boy*. From Charlie's point of view Alice is his *mother*, while Bob is his *father*. Meanwhile, Bob is Alice's *husband*. "Things" in the universe can be classified in various ways, and can even have multiple *labels* attached to them.

Applications of ontology include the *semantic web*, where ontologies form the backbone of data interoperability and intelligent information retrieval. Ontologies enable the integration of diverse data sources, improving the accuracy and relevance of search results. A key standard in this domain is the Resource Description Framework (RDF), which provides a framework for representing information about resources in a graph form. Another

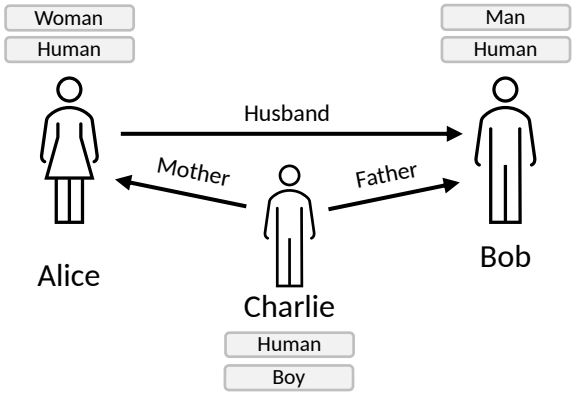


Figure 2.1: Alice, Bob and Charlie classified using an ontology .

important language is the Web Ontology Language (OWL), which is used to explicitly represent the meaning of terms in a vocabulary and the relationships between those terms. OWL extends RDF with more complex and expressive descriptions, supporting rich and nuanced data models necessary for sophisticated reasoning and inference. The real power of ontologies comes when we start processing them to derive new information. SPARQL is an SQL-like query language for querying graph-based data structures, capable of inferring new information from existing data by utilizing the relationships and rules defined in the ontologies. For example, the *child* and *wife* connections can be derived for the example described above (Figure 2.1) using the existing *mother*, *father* and *husband* relations.

2.2 MBSE Languages

In model-based systems engineering (MBSE), models are the primary artifacts of the development process [51], 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 [9].

2.2.1 Systems Modeling Language v2

The Systems Modeling Language v2 (SysML v2) [47] 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 [27] language, with enhanced precision, expressiveness, interoperability, and consistency.

Unlike SysML – which builds upon the Unified Modeling Language (UML) [28] – SysML v2 builds upon the new Kernel Modeling Langauge (KerML) [46]. 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.

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.

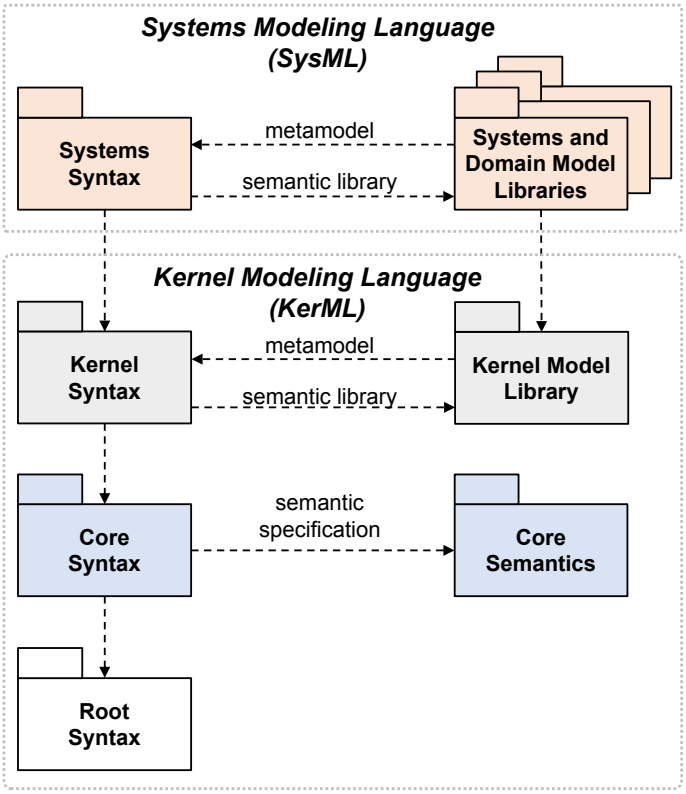


Figure 2.2: An illustration of the KerML and SysML language structure.

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." [46]

Figure 2.2 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.

KerML – and by effect SysML v2 – is ontological, it defines *classifiers* that classify things. The different classifications have various *rules* or *axioms* that govern what labels a given *instance* may have. For example, classifications may be *disjunct*, which ensures no instance can have those labels attached to them.

KerML has *open-world* semantics, unlike many modeling languages. Open-world semantics, as opposed to closed-world semantics, means, that the model does not state explicitly its instances, rather, the axioms *constrain* the possible instances that can exist. Instantiating or interpreting such a language is difficult because one must use *logic solvers* to solve the various constraints and devise a solution satisfying them all.

2.3 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 $M \not\models \gamma$ [13]. 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 2.3), such as the ones used in UPPAAL[1] [39] or Theta[2] [54] 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, giving engineers a chance to understand the error with a step-by-step guide.

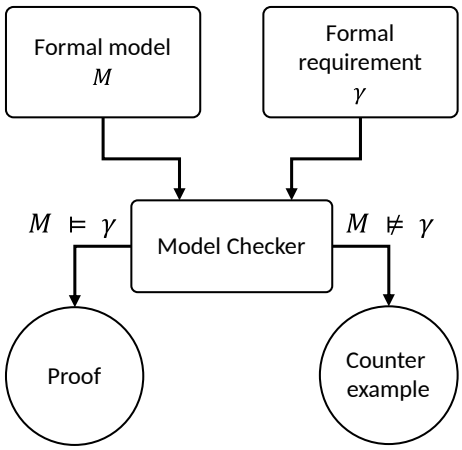


Figure 2.3: An illustration of model checking.

2.4 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 [13, 50, 38, 33, 2, 6, 31, 34]

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

2.4.1 Extended Symbolic Transition System (XSTS)

Extended Symbolic Transition System (XSTS) [24] is an extension of Symbolic Transition System [34] (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 XSTS models as 4-tuples XSTS $= \langle V, Tr, In, En \rangle$ where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of *variables* with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$, e.g. *integer*, *bool* ($\top$ for *true*, $\bot$ for $false$), or *enum*. An *enum* domain is just syntax sugar, a set of *literals* 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., $IV(bool) = \bot$, $IV(integer) = 0$. Every variable v can have a custom initial value $IV(v) \in D_v$ but it is not necessary, because its domain D_v always has one. The *initial state* s_0 is given as the *initial value* for each variable v: $s_0(v) = IV(v)$ if $IV(v)$ exists, otherwise $s_0(v) = IV(D_v)$. The execution of the system starts with assigning the initial value $s_0(v)$ to every variable $v \in V$.

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

XSTS 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.

2.4.2 Theta Model Checking Framework

Theta[3] [54] 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 [32] for the reachability analysis of various formalisms. Theta is capable of processing XSTS models, among many others, and calculating for them either a *Safe* result when the model always respects the specified *invariant expression*, or an *Unsafe* with an execution trace as *wittness* of the invariant violation.

2.5 Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework[4] [43] developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics is an integrated tool aiming 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 2.4.2.

2.5.1 Gamma Behavioral Languages

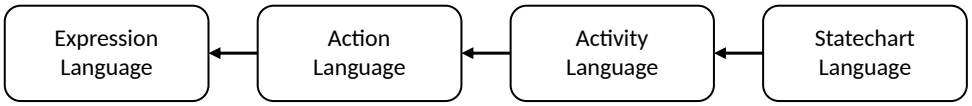


Figure 2.4: The language structure of the Gamma Framework.

Figure 2.4 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)* [57] 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 [61] 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) [22]. Statechart formal semantics provide simple and composite states, entry-exit and doActivity behaviors, orthogonal regions, and transitions with effects.

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

A previous work [53] 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.

```
1  statechart Leader [
2      port control : requires Control
3      port start : provides Start
4  ] {
5      region Main {
6          initial Entry
7          state Idle
8          state Operational
9      }
10     transition from Entry to Idle
11     transition from Idle to Operational
12         when control.fire / raise start.start
13     transition from Operational to Idle
14         when control.stop
15 }
```

Listing 2.1: The leader statechart.

```
1  statechart Worker [
2      port start : requires Start
3  ] {
4      region Main {
5          initial Entry
6          state Idle
7          state Operational
8      }
9      transition from Entry to Idle
10     transition from Idle to Operational
11         when start.start
12 }
```

Listing 2.2: The worker statechart.

2.5.2 Gamma Composition Semantics

As shown in Figure 2.4, 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 [23].

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

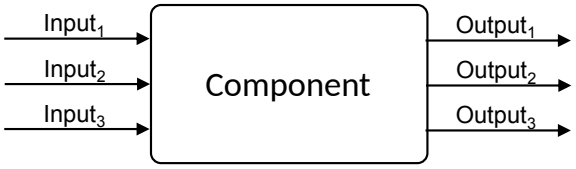


Figure 2.5: Abstract diagram of atomic components.

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 2.5, depicting a component with a set of input and output signals.

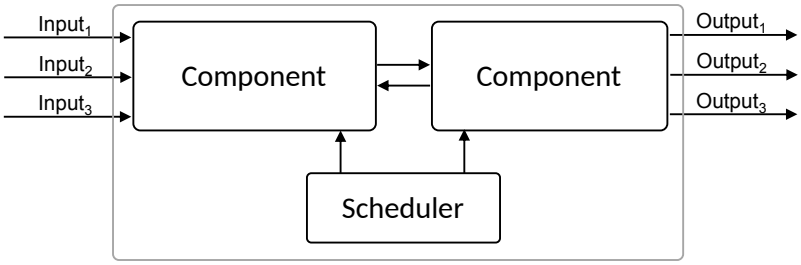


Figure 2.6: Structure of a synchronous composite component.

Synchronous composite components are defined by their internal components and 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 2.6 depicts a generic composite component and its internal structure.

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

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

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 2.7 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 $Input_3$.

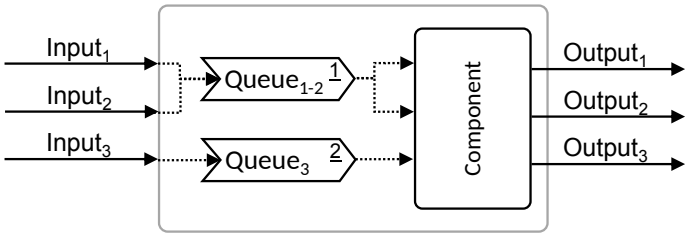


Figure 2.7: Diagram of an asynchronous adapter component.

Asynchronous composite components compose other asynchronous components, connecting them with channels – similarly to synchronous components. Figure 2.8 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.

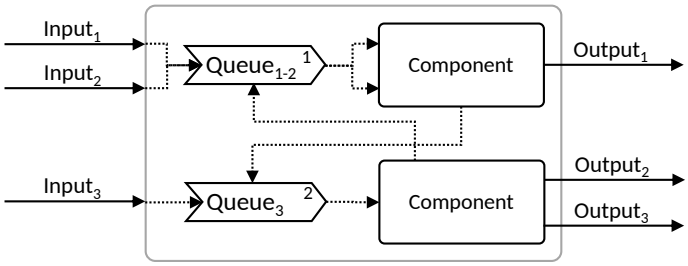


Figure 2.8: Structure of an asynchronous composite component.

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.

```
1 sync System [ port control : requires Control ] {
2     // Instantiating the leader and worker statecharts
3     component leader : Leader
4     component worker : Worker
5     // binding input port
6     bind control -> leader.control
7     // and connecting the leader to the worker
8     channel [ leader.start ] -o)- [ worker.start ]
9 }
```

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

Chapter 3

From Transpilers to Semantic Libraries

Recent trends in systems engineering need formal methods to become widespread across the industry. However, there are several factors hindering developers from creating formal verification tools for the masses. This chapter overviews the direct motivation for this work and provides a new approach for operationalizing high-level engineering languages.

The chapter is structured as follows. Section 3.1 overviews the state-of-the-art related work in the industry, and provides direct motivation for this work. Section 3.2 Introduces a new workflow that aims to solve the problems specified as motivation. Finally, Section 3.3 outlines the exact workflow Semantifyr realizes.

3.1 Motivation and Related Work

This work draws inspiration from current state-of-the-art formal verification tool shortcomings and the recent trend of using ontological behavior models.

3.1.1 Transpilation Using Intermediate Languages

The gap between high-level ontological and low-level formal languages is often bridged by model transformations or transpilation [58, 37, 43, 12]. On top of the syntactic mapping, these model transformations also have to map from a declarative, ontology-based semantics to an operational one. Preserving the semantic equivalence (or at least conformance) of the two representations is crucial for the credibility of the whole analysis process. This is especially challenging when the specification of the semantics of the high-level language is not well-defined, as reported in [18], where the authors analyzed the specification of the PSSM [29] standard, and reported various discrepancies between the specification and its test set, indicating under-specified language semantics.

The process of transforming engineering models into formal, analyzable models is intricate, often necessitating individual effort for each pair of languages. Although attempts have been made to simplify the N×M transformations into N+M utilizing an intermediate language (e.g., [43, 11, 5, 35]), this approach also proves challenging: modifications to the intermediate language are often necessary to support specific high-level languages, making its maintenance difficult.

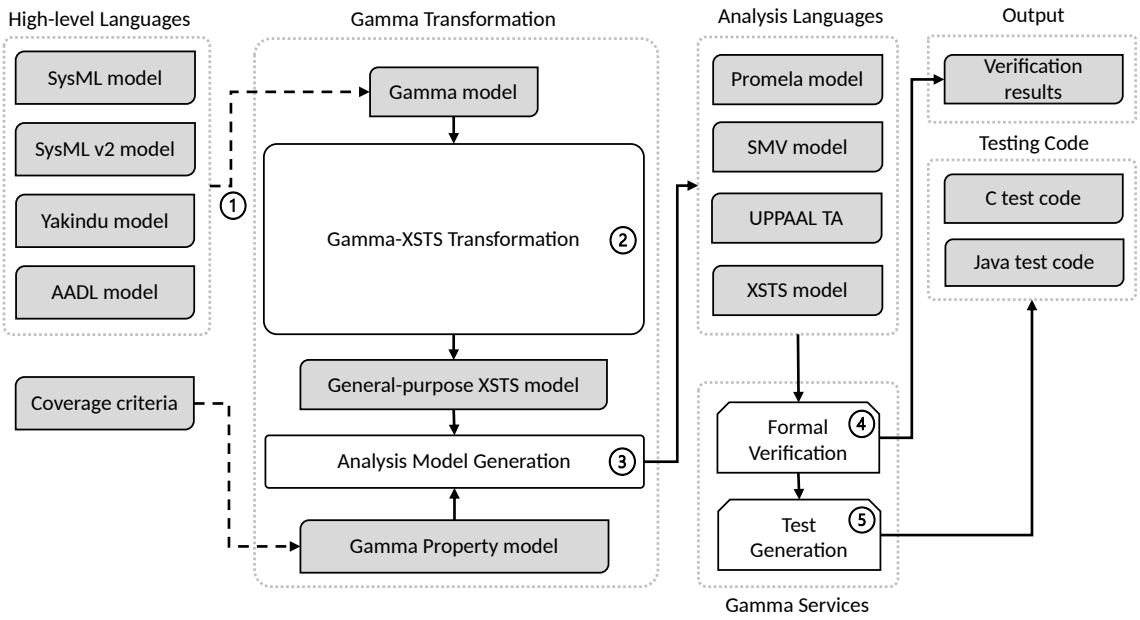


Figure 3.1: The current end-to-end model transformation workflow of the Gamma framework.

For illustration, Figure 3.1 shows the model transformation workflow behind the Gamma tool. Out of the box, Gamma supports Yakindu and SysML v2, however, previous works have used Gamma to verify SysML models as well [35]. 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.

Gamma clearly has several benefits and high-level features, that can help engineers during systems design. However, the approach Gamma uses to transform the models to low-level formalisms proves to be difficult to configure. Supporting new engineering languages often requires intricate modification into the feature set of the Gamma languages – e.g., state machines in SysMLv1 vs SysMLv2 vs Yakindu – resulting in language-specific forks of the transformation, which is not maintainable in the long term. This has been demonstrated in [35], where the authors had to modify the Gamma semantics in various places to support SysML.

3.1.2 Ontological Behavior Modeling

Ontology-based concept modeling (see Section 2.1) is getting increasingly popular in systems engineering. In [60], Lan Yang el. al. provide a thorough overview of the literature regarding the use of ontology in systems engineering. While structural models are commonly ontological, ontological behavior modeling is a relatively new direction, discussed in, e.g., [7, 8]. Ontological behavior models classify 4D traces of a system, encompass-

ing both the structural aspects (3D) and the completely unrolled temporal aspects (4th dimension), as opposed to a 3D model evolving with time. This perspective is the basis of the openCAESAR MBSE framework and methodology developed at JPL [17], as well as the KerML [46] and SysML v2 [47] languages. None of these languages considered execution as a primary aspect, which makes it challenging to analyze these models with simulation or model checking (and other kinds of formal verification techniques building on operational semantics).

To illustrate how ontological behavior modeling works, Figure 3.2 denotes a simple State Machine with two states and a state transition between them. State S1 has an *exit* action associated with it, while S2 has an entry action. Both increment the variable x by one. The transition in between them *accepts* a *toggle* event, in which case it executes the *reset* action, then transitions to the S2 state. Since KerML is defined as an ontology, the specified elements can be decomposed into lower-level concepts defining their meaning in a finer granularity. The decomposition and the corresponding reduction rules are defined in model libraries, building up the standard library from a few fundamental, core elements rooted in first-order logic.

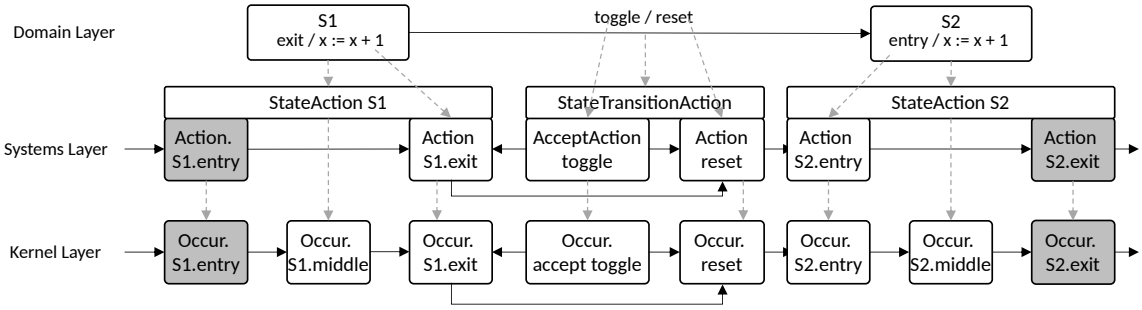


Figure 3.2: SysMLv2 representation of a simple state machine in increasing granularities over the time dimension.

The Systems layer defines *StateDefinitions* and *StateActions*, which model State Machines and States, respectively. States have entry-, do- and exit actions. The transition between the states is a *StateTransitionAction*, which has a *TriggerAction* of an *AcceptAction*, and an *effect* of reset *Action*. In this layer, the arrows denote the ordering between the actions, e.g., *S1 exit* must happen after *toggle accept* and *S1 entry*. Note that the assignment actions are not visualized for the sake of clarity, but would introduce a lot of additional complexity.

In the Kernel layer, Actions are defined as *Occurrences*. Occurrences are things that exist in space and *time*. Temporal relations are defined using *succession* connections between occurrences: succession enforces that instances of the earlier occurrence are finished before the matching later occurrence begins. An execution of this model requires the instantiation of these concepts respecting the relationships and constraints. This will lead to an execution graph, which can be viewed as a refinement of the original model, resolving all the undefined aspects in the open-world semantics into a concrete instance. In this sense, model verification is essentially reasoning about the existence of an instance of the model based on the knowledge base encoded in the model and the libraries.

Ontology-based reasoning and verification have been investigated, e.g., in [17, 10, 3]. While these approaches are sound, they generally do not scale to large models, because ontology reasoners are not designed to handle the size of the knowledge graph that the encoding of a fully unfolded state space would require. Model checking and related techniques have been addressing the state space explosion problem for decades [55, 14], e.g., with abstraction,

symbolic model checking, and compositional verification, often exploiting symmetries or high-level patterns in the models [1, 49, 45, 16]. Contrary to ontology reasoners, model checkers are highly specialized tools optimized to handle huge state spaces in a highly compressed format, while still maintaining the efficiency of reasoning over the state graph. However, these tools generally expect a low-level formal language as input, which usually has (mostly) operational semantics.

3.2 Semantic Libraries

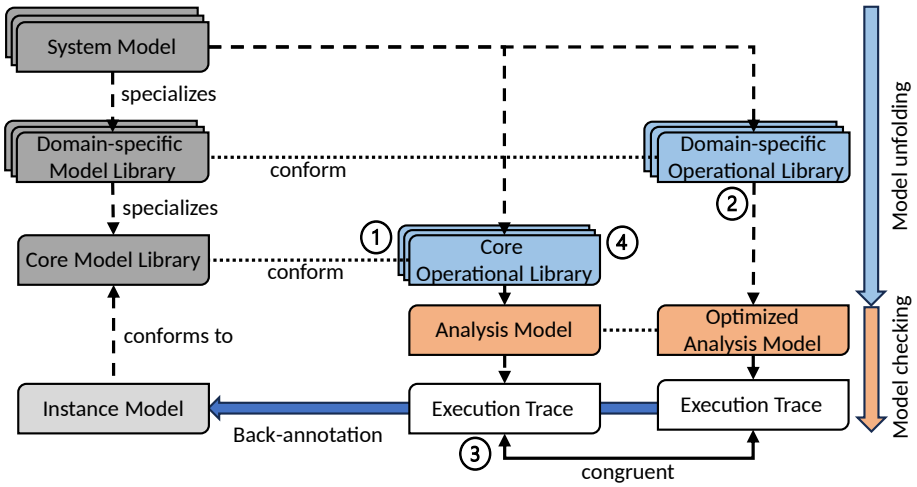


Figure 3.3: Semantic Libraries Overview.

Semantic libraries provide a way to solve the problems with transpilation and ontological behavior models. A high-level overview of the proposed approach is illustrated in Figure 3.3, with descriptions of each part in the following paragraphs.

1) Core Operational Library I propose to extend high-level ontological modeling languages with the ability to specify operational semantics as a first-class citizen. Doing so would allow language designers to specify the exact operational semantics of their language in a Core Operational Library as part of the ontological model library knowledge base. This allows the same model reasoning techniques current ontological modeling languages allow but also provides explicit operational semantics for model execution, simulation, and model checking. System models created in such a language could be directly mapped to the low-level analysis formalism.

2) Domain-specific Operational Library The core operational library contains all the necessary information about the high-level language's semantics. However, there are times when a more abstract representation is desired, e.g., in the state machine domain, the execution of guard expressions or assignments is better left to the verification tool (ontological reasoners would instantiate the execution of operands hierarchically). In such cases, a more abstract domain-specific operational library may be used.

Using such a library results in the analysis model merging some execution traces that would otherwise be separate in the original language. Model checking is thus a simpler problem with such models since the model checker has fewer execution traces to check. Figure 3.4 depicts an example of the execution traces of the example state machine introduced in

Figure 3.2. The top trace corresponds to the domain-specific library, while the lower ones correspond to the Systems and Kernel layers. The lower layers contain more information about the exact unfolding of the behavior, but this information is not necessarily required for the verification. To enforce conformance between the core and domain-specific libraries, the execution traces must be congruent with each other, meaning the domain-specific execution traces must be stutter-equivalent [26] with the execution traces of the core layer.

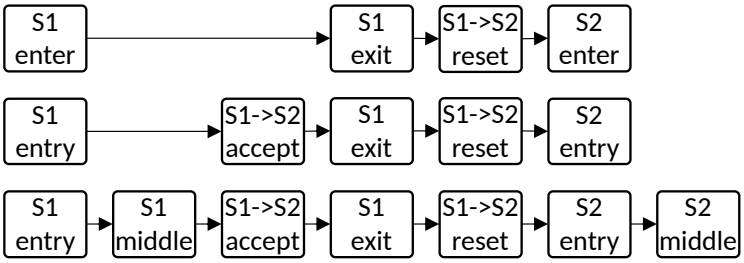


Figure 3.4: Execution traces of the Figure 3.2 State Machine in the Domain, Systems and Kernel layers.

3) Semantic conformance validation To validate the semantic conformance of the core and domain-specific operational libraries, the resulting witness execution trace of a verification process can be mapped back into the original language using back-annotation. Since the original language semantics is axiomatic, checking its validity is as simple as checking if any axiom is violated by the instance model.

4) Semantic variance support With the proposed approach, semantic variances can be supported by implementing multiple variations of core- or operational libraries. Since the libraries are ontologies, the semantic variant libraries could reuse parts from a shared base library. For languages built on top of the same base library (e.g., SysML v2 and a new version of AADL [19] are both based on KerML), analysis models may interface with each other along the common base, allowing multi-domain verification with heterogeneous models.

3.2.1 Semantic Library Using KerML

A straightforward way to realize the proposed vision could be to model the low-level analysis model directly in KerML, and trace specific parts of the language semantics to instances of the formalism. As illustration, Listing 3.1 depicts an excerpt of a possible XSTS metamodel defined in KerML, that could be used to trace the semantics of KerML (and transitively SysMLv2) types. To avoid the problems with the 4D space-time, it is essential to not use *Occurrences* and *Successions*, as that implies a temporal dimension.

Unfortunately, since the SysML/KerML pilot implementation is not yet finished, there is no working model reasoner capable of producing valid instances of KerML models. Thus, it is currently impossible to prototype such a solution using KerML.

```
classifier Variable;
abstract classifier Operation;
classifier AssignmentOperation :> Operation; // ...
classifier AssumptionOperation :> Operation; // ...
classifier HavocOperation :> Operation { feature variable : Variable[1..1]; }
classifier CompositeOperation :> Operation { feature operations : Operation[0..*]; }
classifier SequenceOperation :> CompositeOperation;
classifier ChoiceOperation :> CompositeOperation;
classifier Transition { feature operations : Operation [0..*]; }
```

Listing 3.1: A possible XSTS metamodel in KerML.

3.3 Semantifyr Workflow

OXSTS is a new ontological-operational language that mimicks the way KerML uses classifiers, however, it has closed-world semantics as opposed to KerML's open-world semantics, which makes its instantiation simpler. Semantifyr is a proof-of-concept tool capable of instantiating OXSTS models and unfolding them into XSTS, mapping them directly to the operational analysis formalism.

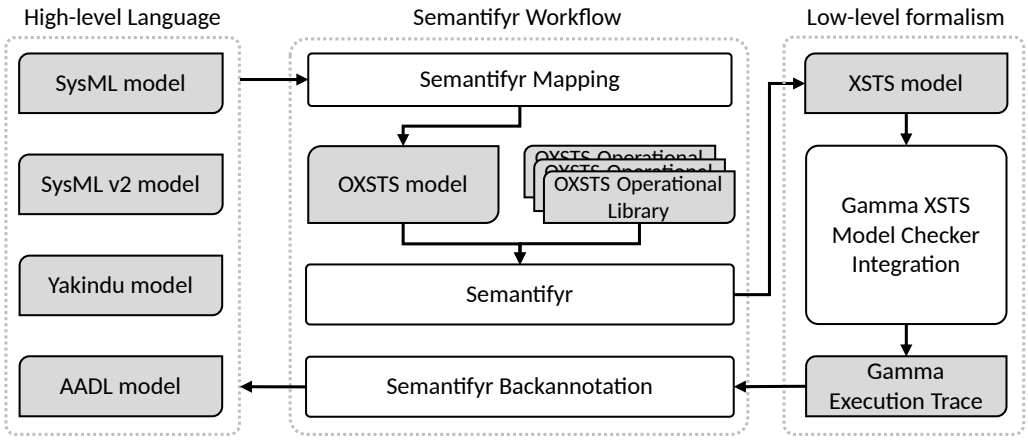


Figure 3.5: The proposed end-to-end model transformation workflow using Semantifyr.

The Semantifyr workflow depicted in Figure 3.5 is a simplified version of the vision defined in Figure 3.3. By modeling the various language constructs available in the high-level engineering language (e.g., in KerML/SysML), a semantic mapping is possible between the language constructs and the types defined in the operational library. Each modeling language (and semantic interpretation variant) has a corresponding OXSTS operational library, thus the resulting OXSTS model is a mapping of the original model to an OXSTS model importing and reusing the necessary operational library types (such as *States*, *Transitions*, *Actions*). The constructed OXSTS models are unfolded into the XSTS formalism by Semantifyr. The resulting XSTS model is then passed through a model checker using the Gamma integration, after which the execution traces can be back-annotated to the original high-level language to describe them in the context of the original high-level engineering model.

Chapter 4

Semantifyr

Semantifyr is an open-source tool developed to support the declarative definition of engineering model semantics in a highly configurable manner. It uses the OXSTS language, a new ontological-operational modeling language capable of modeling high-level constructs in a declarative manner. By specifying semantic libraries in OXSTS, Semantifyr allows the operationalization of high-level languages.

This chapter is structured as follows. First, Section 4.1 introduces the new OXSTS language and its features. Next, Section 4.2 details various conformance tests for the language using hand-crafted models. Finally, Section 4.3 details the implementation of Semantifyr.

4.1 Objective XSTS

Objective XSTS is a new language implementing several meta-programming features known across the industry (see [40]), built on top of the XSTS [24] analysis formalism. The language is made up of two parts. The first part supports ontological modeling similar to KerML, while the second grounds the semantics with composable operational atoms. Essentially, the ontological universe of the language is the set of all atomic XSTS statements, and the valid models are the statements resulting from the composition of the atoms. This section provides an overview of the various parts using descriptions and excerpts from the Gamma Semantic Library introduced in Chapter 5. Table 4.1 depicts a summary and description of all available language features.

4.1.1 Ontological Part

Types and Features **Types** act as classifications for the various *instances* that can be modeled by an OXSTS model. Types can specialize each other.

Types – and instances of types – may reference each other using **Features**. Features have *multiplicities*, specifying the cardinality of their instances. Features come in several kinds. **Containments** define compositional associations between two types. **References** are not compositional, and can provide the crosslinks between instances. **Derived references** are calculated during the instantiation of the model, and are specified by **graph patterns**. The language supports ontological modeling techniques, such as feature *subsetting* and *redefinition* and usage-oriented modeling.

There are two built-in special types: *Integer* and *Boolean*. They are special since their domain is either the infinite set of all integers $\mathbb{Z}$, or the set $B = \{true, false\}$.

Category	Feature	Description
Linking	Package	Packages structure code into logical groups.
	Import	Imports elements from other packages.
Type	Enum	Custom enumeration type.
	Custom	Custom types with features, transitions.
	Target	Special entry-point type.
Inheritance	Feature redefinition	Specializing inherited features in child types.
	Transition redefinition	Specializing inherited transitions in child types.
	Feature subsetting	Feature decomposition.
Variable	Data-typed	Integer or Boolean variables.
	Feature-typed	Points to instances in the feature.
	Enum typed	Custom enumeration type.
Feature	Data-typed	Features that represent data.
	Instance-typed	Features that represent instances.
	Containment	Features specifying containment hierarchy.
	Reference binding	References bound to specific instances.
	Derived reference	References derived from Patterns
Pattern	Type constraint	Custom type constraint for instances.
	Feature constraint	Association constraint between instances.
	Pattern constraint	Constrain instances using other patterns.
Operation	Assumption	Assume the expression is true.
	Assignment	Set the variable to the expression.
	Havoc	Randomize the value of the variable.
	Sequence	Execute internal operations in order.
	Choice	Choose a branch to be executed.
	If-else	Choose a branch using a guard expression.
Inline op.	Inline call	Inline the body of a transition to the call-site.
	Inline sequence	Inline transition of instances into a sequence.
	Inline choice	Inline transition of instances into a choice.
	Inline if-else	Conditional inlining.

Table 4.1: OXSTS langauge features.

```
1  type Event
2  type Component {
3      feature events : Event [0..*]
4      containment inputEvents :> events : Event[0..*]
5      containment outputEvents :> events : Event[0..*]
6  }
7  type Action
8  type RaiseEventAction : Action
9  containment a :> actions : RaiseEventAction {
10     reference ::> event : Event = toggleEventA
11 }
```

Listing 4.1: Type and feature example model. Depicts an excerpt from the Gamma Semantic Library and a special *RaiseEventAction* redefining its *event* reference to point to *toggleEventA*.

Listing 4.1 depicts an excerpt model from the Gamma Semantic Library, and a containment specialization *t* that redefines its *to* feature. The component type may contain several *events*. Input- and output events are stored in separate features, which subset the *events* feature. Feature redefinition is used to specialize the specific feature in specific contexts.

Patterns Graph patterns provide a way to specify complex reference rules that are evaluated automatically during instantiation, instead of requiring the modeler to set them manually. The current implementation uses the time-proven Viatra Query Language [4], however, supporting additional query languages (such as SPARQL) would be a simple matter of extending the language and integrating a query interpreter engine.

The VQL pattern language is integrated in such a way as to allow the referencing of types and their features directly. The derived features must use patterns that have exactly two parameters: the "self" instance of the context, and the output value. The following VQL features are supported: type constraints, feature constraints, negation of constraints, and pattern constraints with $*$ and $+$ transitive closures. See [4] for more details on VQL.

```
1  pattern parentState(childState: State, parentState: State) {
2      State.regions(parentState, parentRegion)
3      Region.states(parentRegion, childState)
4  }
5  pattern parentRegion(childState: State, parentRegion: Region) {
6      Region.states(parentRegion, childState)
7  }
8  type State {
9      derived reference parent : Region[0..1] as pattern parentRegion
10     derived reference parentState : State[0..1] as pattern parentState
11     containment regions : Region[0..*]
12 }
```

Listing 4.2: An excerpt of the State type from the Gamma Semantic Library.

Listing 4.2 depicts the State type excerpt from the Gamma Semantic Library. The State type has *parent* and *parentState* references to a *Region* and *State* respectively. To reduce user error, these references are defined as *derived references* using *Patterns*. The Pattern *parentRegion* connects the "self" *childState* to its *parentRegion* by specifying, that the *childState* must be in the *parentRegion*'s *states* feature. The *parentState* pattern is specified in a similar fashion, only extending it with one more indirection using the *regions* feature of the *State* type.

4.1.2 Operational Part

XSTS is used as the base language to operationalize the language semantics. XSTS models can be reduced to SMT [2] formulae, but the language allows direct execution as well via its statements.

XSTS summary XSTS supports **variables** with various domains, **assignments**, **non-deterministic assignments**, and **assumptions** to constrain variables. Operations are atomic. Composing operations can be achieved with **choices**, **sequences** and **decisions**. Choices choose non-deterministically from their branches, while sequences execute all operations in the given order. Decisions simply choose either of the two branches depending upon the guard expression. Note, that OXSTS does not have *env* transitions, since they

can easily be emulated by placing them at the end of the main *tran* transition. See Section 2.4.1 for more information about XSTS.

XSTS extension 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 2.4.1), 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 automatically transformed back to XSTS. Listing 4.1.1 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 4.1.1: Choice-else using classic XSTS (left) and the new else syntax (right).

Formally, a choice of form $op_1 \dots op_n$, op_{else} means a non-deterministic choice between operations (branches) $op_1, \dots, op_n$ where $op_i \in Ops$ with an else branch $op_{else} \in Ops$. This means that exactly one executable branch op_i or op_{else} will be executed. A branch $op = (s, s')$ can not be executed if $s' = \emptyset$, 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 $1 \leq i \leq n$), then choice's successor state is s'_{else}. Generally, the set of successor states is the union of the results of the branches $\bigcup_{i=0}^{n} s'_i$ if $\exists(s'_i \neq \emptyset)$, otherwise s'_{else}.

Example Listing 4.3 depicts an excerpt of the *Timeout* type from the Gamma Semantic Library. Timeouts keep track of time using a variable called *remainingTime*. To allow specializing the duration of the timeout, a *deltaTime* reference is defined, which is an *Integer*. Timeouts work by first checking whether they are *elapsed* (remaining time is less than zero), then decrease the remaining time with the specified delta time. Timeouts also provide a transition to check whether they are up using their *isUp* transition, which places an assumption over the remaining time variable.

4.1.3 Combining the Parts

To extend the ontological language with operational semantics, I used constructs from metaprogramming to manipulate the atomic XSTS elements by composing them in various ways in the context of types. Types and features can define variables and transitions. With the introduction of **inline operations**, the model can reference transitions to be unfolded by inlining their content at the call site. Composition is supported by **composite**

```
type Timeout {
    var remainingTime : Integer = 0
    reference deltaTime : Integer = 1
    tran {
        if (remainingTime > 0) { remainingTime := remainingTime − deltaTime }
    }
    tran isUp { assume (remainingTime <= 0) }
}
```

Listing 4.3: An excerpt of the Timout type from the Gamma Semantic Library.

inline transitions, which unfold into simple inline transitions placed into the appropriate composite transition: choice or sequence. Transitions can be redefined, similarly to features. Static decision points can be modeled with **inline if** operations, that inline their bodies conditionally. Finally, **feature-typed** variables support the runtime tracking of instances: their domain is the set of instances in the specified feature.

```
type Region {
 containment states : State[0..*]
 ctrl var activeState : states [0..1] = Nothing
}
type State {
 containment regions : Region[0..*]
 containment exitActions : Action[0..*]
 containment entryActions : Action[0..*]
 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
 }
}
```

Listing 4.4: An excerpt of the Region and State types from the Gamma Semantic Library.

Listing 4.4 depicts an excerpt of the *Region* and *State* types from the Gamma Semantic Library. Regions contain states, which are tracked using a *feature-typed* variable called *activeState*, which is initially *Nothing*. States may also contain *regions* (sub-states, parallel states), and *exit* actions. Exiting is defined using static recursion: the state first recursively exits the common region, then recursively deactivates its internal regions. Recursive exiting is done by first inlining its exit actions, setting its parent region's active state to *Nothing*, then continuing with its parent state if the recursion has not reached the common

region. State entering is implemented in a similar fashion, except the order is changed, and the parent region's active state is set to the current *State*.

Target definitions function as entry points for Semantifyr, determining what to instantiate from the model. Targets define the *universe* of a given verification, such as a specific system configuration. It determines the *init*, *env*, and *tran* transitions (see Section 2.4.1) of the resulting XSTS model as well as its variables.

```
target Mission {
    containment component : Component
    init { inline component.init() }
    tran {
        inline component.main()
        inline component.passTime()
    }
}
```

Listing 4.5: An example Target definition containing a component.

Listing 4.5 depicts a simple *target* definition containing a single *Component*. It specifies that a single *Component* instance must be constructed (along with its children depending upon its containment features). Its init transition is the *component*'s init inlined, while its main transition is first the main of the component, next its *passTime* transition.

4.2 Conformance Tests

For feature completeness and validation, I constructed a conformance test suite for OXSTS. The primary goal of the suite is to provide a complete coverage of all OXSTS language features. The test cases are made up of an OXSTS target and an expected XSTS model. The test suite runs Semantifyr on each target definition, transforms it into XSTS, and then compares it to the expected XSTS model. A case is considered passing *iff* the expected and the resulting models are textually equivalent. This is possible since Semantifyr is deterministic. Listing 4.2.1 depicts one of the test cases. The left side contains the Target and model definitions, while the right side contains the expected output.

```
type Atom
type Holder {
    feature atom : Atom
    var current : atom[0..1] = Nothing
}
target Mission {
    containment holder : Holder {
        containment a :> atom : Atom
        containment b :> atom : Atom
        containment c :> atom : Atom
    }
    init { holder.current := Nothing }
    tran { holder.current := holder.b }
}
```

```
type ___holder___atom___type : {
    ___holder___a___literal,
    ___holder___b___literal,
    ___holder___c___literal,
    ___Nothing___literal
}
var ___holder___current : ___holder___atom___type
    = ___Nothing___literal
trans {
    ___holder___current := ___holder___b___literal;
}
init {
    ___holder___current := ___Nothing___literal;
}
env {}
```

Listing 4.2.1: "Variable with Nothing expression" test case from the OXSTS conformance test suite.

The test suite (Table 4.2) contains 55 hand-crafted test models each testing specific features of the language. Note, that all features have at least one case that covers it primarily, and many cases that cover them secondarily. All implemented conformance tests successfully pass. These tests not only confirmed Semantifyr's correct operation but also helped during development by enabling test-driven development.

	Linking	Enum	Type	Target	Feature redefinition	Transition redefinition	Feature subsetting	Variable	Feature	Pattern	Operation	Inline operation	**# of test models**
Feature				◐	◐		●		◐		◐	◐	4
Feature Typing			●	◐				●	●		◐	◐	6
Import	●												1
Inheritance			●	◐	●	●	◐	●	◐		◐	◐	7
Operations				◐				◐			●		6
Inline Operations				◐							◐	●	17
Instance		●	●	◐			●	◐	●		◐	◐	9
Pattern			◐	◐				●	●	●	◐	◐	1
Target				●				●					4
Sum													55

Table 4.2: Automated conformance test cases testing specific features. ◐ means a secondary coverage, while ● means a primary feature coverage.

4.3 Implementation

In the past decades, Eclipse 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 or Theia.

To adapt to this change, the focus was on enhancing Semantifyr's versatility across different IDEs 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 build system, offering automation and efficiency. Semantifyr is implemented in Kotlin, a modern JVM-based language known for its advanced features and great interoperability with other JVM-based languages, such as Java. Semantifyr is free and open source, available[1] under the EPL v2 license.

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://github.com/ftsrg/semantifyr`

Traditionally, compilers exhibit five phases: lexical analysis & syntactical analysis, model validation, semantic analysis, optimization, and code generation [59]. This is no different with Semantifyr. Figure 4.1 depicts the five main phases of the OXSTS → XSTS transformation pipeline.

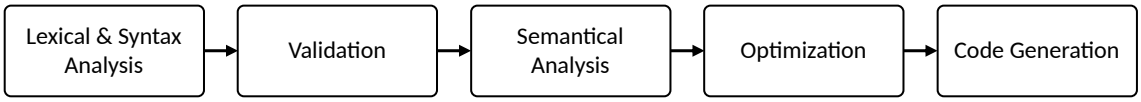


Figure 4.1: The main transformation phases of Semantifyr.

4.3.1 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. Semantifyr uses EMF and Xtext to realize this phase.

Eclipse Modeling Framework Eclipse EMF 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, such as Viatra [4].

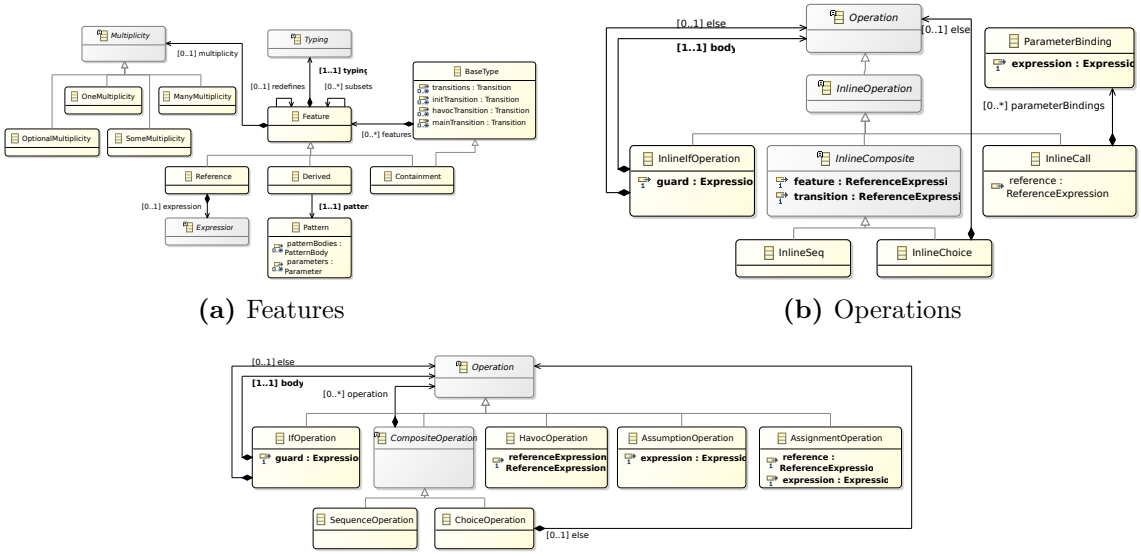
Figure 4.2 depicts a small excerpt from the metamodel of the OXSTS language, defining the structure of features, operations, and inline operations.

Xtext Eclipse Xtext is a framework for developing programming and domain-specific languages. It covers all aspects of a complete language infrastructure, from parsers, through linkers, to compilers. Xtext provides the means to create Eclipse-first 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 or Theia.

4.3.2 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. Various validation rules have been implemented to ensure only semantically legal models are transformed and to provide the users with valuable insights about incorrect models.

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



(a) Features **(b)** Operations

(c) Inline Operations

Figure 4.2: An excerpt from the OXSTS metamodel in Ecore format.

The following validation rules are implemented.

- Redefinition or subsetting does not respect specialization rules.
- Transition inlining defines too many parameters.
- Transition inlining does not define all parameters.
- Incorrect transition inlining parameter type.
- Composite transition inlining must reference a feature with *many* multiplicity.

4.3.3 Semantical Analysis

The semantical analysis phase processes the syntactically and semantically legal model, transforming all meta-programming constructs into plain XSTS constructs. The semantical analysis is done in three steps, as depicted in Figure 4.3.

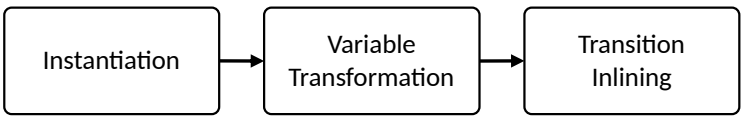


Figure 4.3: The main steps of the Semantical Analysis phase.

Instantiation

First, the specific Target definition is instantiated. Instantiation entails the collection of all *containment* features in the target with strict multiplicity and the construction of instances for each containment. The internal containments are also instantiated during this process, resulting in an instance tree. When the complete instance tree is created the reference bindings are resolved.

There are two kinds of reference bindings. In the case of a simple reference, the binding expression is resolved in the context of the instance, and the resulting instances are placed in the correct feature slots. In the case of derived references, the pattern is first transformed and then executed. Since the language offers an integrated version of the Viatra Query Language, it first must be compiled to VQL. Each Type and Feature access is replaced with a special built-in pattern called *instanceOfType* and *instanceInFeature* respectively.

```
1  pattern OXSTS____instanceOfType(instance: Instance, typeName: java String) {
2      Instance.containment(instance, containment);
3      find OXSTS____featureType(containment, type);
4      Type.supertype*(type, actualType);
5      Type.name(actualType, typeName);
6  }
7  pattern OXSTS____instanceInFeature(container: Instance, held: Instance,
8   typeName: java String, featureName: java String) {
9      find OXSTS____instanceOfType(container, typeName);
10     Instance.associations(container, association);
11     Association.feature(association, feature);
12     Feature.name(feature, featureName);
13     Association.instances(association, held);
14 }
15 pattern Statecharts___parentRegion(childState, parentRegion) {
16     find OXSTS____instanceOfType(childState, "State");
17     find OXSTS____instanceOfType(parentRegion, "Region");
18     find OXSTS____instanceInFeature(parentRegion, childState, "Region", "states");
19 }
```

Listing 4.6: VQL representation of the *parentRegion* pattern, and an excerpt of the OXSTS VQL library.

Listing 4.6 depicts the OXSTS helper patterns and the VQL version of the *parentRegion* pattern defined in Listing 4.2. The *childState* and *parentRegion* are both constrained with their respective types, and a feature constraint is placed ensuring the *childState* is in the *states* feature of the *parentRegion* instance. *OXSTS_instanceOfType* works in the following fashion. It only matches, when the named type is in the ancestry of the instance's declared type. First, the declared type of the instance is found using its containment (the containment along which it was instantiated) feature. Next, the type ancestry is calculated using the * transitive closure, which allows the *actualType* variable to equal *type* itself; any type is inside its type ancestry. Finally, the pattern asserts that the found *actualType*'s name must be *typeName*. Similarly, *OXSTS_instanceInFeature* checks that there is indeed an *association* that holds the *instance* in the *container* with the required *featureName* name.

Using the generated VQL patterns, the VQL interpreter is called and the patterns are executed in the context of the instance. The resulting instances are then placed in the specified *derived reference*.

At the end of this process the *root instance* is returned, which represents the whole instance tree. At this point, all features and references are known for each instance, thus the model is considered fully instantiated.

Variable Transformation

This step constructs and transforms the declared variables of the created instances. Feature-typed variables must be transformed into simple variables with enumeration defi-

nitions. For each feature-typed variable, the instances contained in the feature are collected and turned into an *enumeration* type with a literal for each instance. If the variable is optional, then a special *nothing* literal is also added. Since all features and references are known at this point, the enumeration literals can simply be constructed by the associated instances. See the previously introduced Listing 4.2.1 for an example.

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, taking care to properly rewrite their context, possibly resulting 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: the XSTS model has been constructed, and the Target is considered fully *unfolded*.

4.3.4 Optimization

The fully unfolded model now contains only XSTS constructs, however, it may contain redundant elements, such as choices with a single branch, sequences inside sequences, or always false expressions. The optimization step simplifies the model using semantic-preserving model transformations. The optimizations are grouped into two categories: redundant operation removals, and expression evaluation.

Redundant operation optimizations:

- Single branch *choices* can be replaced by their single branches;
- Empty *sequences*, *choices* or *ifs* can be removed;
- Redundant *choice-else* branch can be removed;
- *Sequences* and *choices* can be flattened.
- Removal of *true* assumptions;
- Propagation of *false* assumptions;

Expression evaluation optimizations:

- *Or* expressions with a *true* operand, and *and* expressions with a *false* operand can be replaced with their other operands;
- Constant evaluable expressions can be compile-time evaluated.

Gamma 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. Similarly, Theta also supports various runtime optimizations. However, the optimization step of Semantifyr is not redundant: they focus on OXSTS-specific optimizations, that could be difficult to implement without OXSTS domain knowledge – i.e., without the original OXSTS model. These optimizations also significantly reduce the size

of the model – on average models become ~ 20 times smaller after removing redundant elements – which helps during conformance testing and manual readability of the resulting models.

4.3.5 Code Generation

Finally, during the code generation phase, the now optimized XSTS model is serialized into its textual representation. The code generator can output XSTS with choice-else branches rewritten into simple branches. An equivalent form of the else branch is only allowed to be executed *iff* all other branches are disallowed. 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 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* or *assignment* operation is `assume (true)`.
- Assumption of a *sequence* or *choice* operation is the conjugation or disjunction of its inner operations' assumptions, respectively.
- Assumption of an *if* operation is the following: $(guard \wedge body) \vee (\neg guard \wedge else)$, where *guard*, *body*, and *else* are assumptions from the operation.

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 valid if the assumptions do not reference variables that are set inside the operation itself, like in Listing 4.7. In such cases, the relevant subset of the operation must be placed into the *else* branch, in order to simulate that behavior. However, such modeling patterns are not common, they could be detected with validation rules, and could even be optimized away using constant-propagation techniques.

```
1 choice {
2     assume (a == 10)
3     a := 20
4     assume (a < 30) // this assumption depends on the overwritten value of a
5 }
```

Listing 4.7: Simple example of a choice branch with an assumption referencing the internally overwritten variable "a".

Chapter 5

Gamma Semantic Library

To support the semantics of existing Gamma models specified with statecharts and composition (see Section 2.5) 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 5.1, and showcases its usage through an example model in Section 5.2.

5.1 Semantic Library

The most basic type is the *Event* type. Events are represented with a single boolean flag, as shown in Listing 5.1. The Event type exposes various transitions for easy manipulation, e.g., the *isSet* transition with the *isActive* assumption.

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

Listing 5.1: The Event type in the Gamma Semantic Library.

```
type Timeout {
    var remainingTime : Integer = 0
    reference deltaTime : Integer = 1
    tran {
        if (remainingTime > 0) {
            remainingTime :=
             remainingTime − deltaTime
        }
    }
    tran isUp {
        assume (remainingTime <= 0)
    }
}
```

Listing 5.2: The Timeout type in the Gamma Semantic Library.

Timeouts (depicted in Listing 5.2) are modeled using an integer variable that is counting down to 0. To prime the timeout, one has to set the *remainingTime* variable to any value. The *main* transition decreases the variable by *deltaTime* until it reaches 0. The timeout is considered *up* when the timer variable equals zero.

Events and Timeouts are primarily used by Triggers. Triggers (see Listing 5.3), 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 { tran isTriggered { } }
type EventTrigger : Trigger {
    reference event : Event[1..1]
    tran isTriggered {
        inline event.isSet()
    }
}
type TimeoutTrigger : Trigger {
    reference timeout : Timeout[1..1]
    tran isTriggered {
        inline timeout.isUp()
    }
}
```

Listing 5.3: The Trigger type in the Gamma Semantic Library.

```
type Action
type RaiseEventAction : Action {
    reference event : Event[1..1]
    tran {
        inline event.set()
    }
}
type SetTimeoutAction : Action {
    reference timeout : Timeout[1..1]
    reference amount : Integer = 1
    tran {
        timeout.remainingTime := amount
    }
}
```

Listing 5.4: 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 5.4 defines the *Action* type and its sub-types: *RaiseEventAction* and *SetTimeoutAction*. Note, that *SetTimeoutAction* uses an *amount* feature to set the timeout. This allows us to redefine the exact time value of the action at use site.

```
type Component {
 feature events : Event[0..*]
 feature inputEvents:>events:Event[0..*]
 feature outputEvents:>events:Event[0..*]
 feature timeouts : Timeout[0..*]
 tran resetInputEvents {
  inline seq inputEvents -> reset
 }
 tran resetOutputEvents {
  inline seq outputEvents -> reset
 }
 tran passTime {
  inline seq timeouts -> main
} }
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..*]
 tran passTime {
  inline seq timeouts -> main
  inline seq components -> passTime
 }
}
type SyncComponent : CompositeComponent {
 init {
  inline seq components -> init
 }
 tran {
  inline resetOutputEvents()
  inline seq components -> main
  inline seq channels -> main
  inline resetInputEvents()
} }
```

Listing 5.1.1: The Component type hierarchy in the Gamma Semantic Library.

Components represent the central elements of the composition language in Gamma (see Listing 5.1.1). Components contain events – categorized into *input* and *output* events – and timeouts. Each component comes with input-output event reset transitions, and a *passTime* transition that runs its timeouts. *Channels* are modeled as a simple event forwarding type, proxying the *inputEvent* to the *outputEvent*. *CompositeComponents* may contain other components and channels. Composite components inline their internal com-

ponent's *passTime* transition. 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 semantics is conformant with the semantics defined in Section 2.5.

Statecharts are special *Components* defined in Listing 5.5. 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.

```
1 type Statechart : Component {
2  feature regions : Region [0..*]
3  init { inline seq regions -> activateRecursive }
4  tran {
5   inline resetOutputEvents()
6   choice { inline seq regions -> fireTransitions } else { }
7   inline resetInputEvents()
8 } }
```

Listing 5.5: The Statechart type in the Gamma Semantic Library.

The *Region* type is defined in Listing 5.6. 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).

```
1  type Region {
2   feature states : State [0..*]
3   feature abstractTransitions : AbstractTransition [0..*]
4   feature transitions :> abstractTransitions : Transition [0..*]
5   feature entryTransitions :> abstractTransitions : EntryTransition [0..*]
6   ctrl var activeState : states [0..1] := Nothing
7   tran activateRecursive { inline seq entryTransitions -> main }
8   tran deactivateRecursive { inline seq states -> deactivateRecursive }
9   tran fireTransitions { inline choice transitions -> main else {
10   inline choice states -> fireTransitions else { }
11 } } }
```

Listing 5.6: The Region type in the Gamma Semantic Library.

Transitions refine the *AbstractTransition* type. The *EntryTransition* is a special transition that does not have a source state. The *commonRegion* of the transitions is defined using a Pattern, similarly to the parent state feature. The *Transition* type is defined in Listing 5.1.2. Transitions contain a single trigger, guards, 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.

States implement most of the state-transition behavior of Gamma statecharts. As shown in Listing 5.7, the *State* type references the parent region and the parent state. The *deactivateRecursive* transition is called when an upper state is left, since in such cases the lower states must be exited as well. Transition firing works by first checking, whether the state has inner regions or not. If it does not, then it inlines a *false* assumption, since in that case it is impossible to execute its inner transitions. This ensures the model does not allow losing input events. Otherwise, an *isActive* assumption is placed, ensuring this *composite* state is active, and then the inner regions' transitions are placed into a choice,

```
pattern transitionParentRegion(
  transition : AbstractTransition, region : Region
) {
 Region.transitions(region, transition)
}
type AbstractTransition {
 derived reference commonRegion :
   Region[1..1] as pattern
   transitionParentRegion
 reference to : State[1..1]
}
type EntryTransition : AbstractTransition {
 tran { inline to.enter(commonRegion) }
}
```

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

Listing 5.1.2: The Transition type hierarchy in the Gamma Semantic Library.

with an empty else. The else is required, since the statechart may be in a state where there are no transitions that are triggered by the incoming events.

```
type State {
 derived reference parent : Region[0..1] as pattern parentRegion
 derived reference parentState : State[0..1] as pattern parentState
 tran isActive { assume (parent.activeState == Self) }
 tran deactivateRecursive {
  inline seq regions -> deactivateRecursive
  if (parent.activeState == Self) {
   parent.activeState := Nothing
   inline seq exitActions -> main
 } }
 tran fireTransitions {
  inline if (regions == Nothing) { assume (false) } else {
   inline isActive()
   choice { inline seq regions -> fireTransitions } else { }
} } }
```

Listing 5.7: Excerpt of the State type in the Gamma Semantic Library. Only those parts are shown which were not included in Listing 4.4.

5.2 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 5.8 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. The worker statechart, depicted in Listing 5.9, is modeled similarly. The composition of the system should be a *synchron composite* component (see Listing 2.3). Depicted in Listing 5.10, it composes the two statecharts by refining the *SyncComponent*

type, creating two containments subsetting the *components* feature, and creating a channel between the two statecharts.

Finally, Listing 5.11 defines the verification environment of the System component. First, the instantiated system is initialized. The main transition simulates the environment by *havocing* the two input events 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 2.4.2) processes the unfolded XSTS model, and can 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 is checked by stating the state is never active.

```
type LeaderStatechart : Statechart {
 containment fireEvent :> inputEvents : Event
 containment stopEvent :> inputEvents : Event
 containment startEvent :> outputEvents : Event
 containment Main :> regions : Region {
  containment et1 :> entryTransitions : EntryTransition { reference ::> to : State = Idle }
  containment Idle :> states : State
  containment idleToOperational :> transitions : Transition {
   reference ::> from : State = Idle
   reference ::> to : State = Operational
   containment t :> trigger : EventTrigger { reference ::> event : Event = fireEvent }
   containment a :> actions : RaiseEventAction { reference ::> event : Event = startEvent }
  }
  containment operationalToIdle :> transitions : Transition {
   reference ::> from : State = Operational
   reference ::> to : State = Idle
   containment t :> trigger : EventTrigger { reference ::> event : Event = stopEvent }
  }
  containment Operational :> states : State
 }
}
```

Listing 5.8: The OXSTS version of the Leader statechart defined in Listing 2.1.

```
type WorkerStatechart : Statechart {
 containment startEvent :> inputEvents : Event
 containment Main :> regions : Region {
  containment et :> entryTransitions : EntryTransition { reference ::> to : State = Idle }
  containment Idle :> states : State
  containment idleToOperational :> transitions : Transition {
   reference ::> from : State = Idle
   reference ::> to : State = Operational
   containment t :> trigger : EventTrigger { reference ::> event : Event = startEvent }
  }
  containment Operational :> states : State
 }
}
```

Listing 5.9: The OXSTS version of the Worker statechart defined in Listing 2.2.

```
type System : SyncComponent {
 containment leader :> components : LeaderStatechart
 containment worker :> components : WorkerStatechart
 containment startChannel :> channels : Channel {
  reference ::> inputEvent : Event = leader.startEvent
  reference ::> outputEvent : Event = worker.startEvent
 }
}
```

Listing 5.10: The OXSTS version of the synchronous System defined in Listing 2.3.

```
target Mission {
 containment 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 {
  system.worker.Main.activeState != system.worker.Main.Operational
 }
}
```

Listing 5.11: The target definition specifying the mission environment of Listing 5.10.

Chapter 6

Evaluation

To evaluate the presented approach and the constructed Gamma Semantic Library, I modeled two high-level engineering models using the OXSTS langauge, and verified them using Semantifyr and Theta. By comparing the output of the Theta model checker for both the Semantifyr and Gamma versions of the models, the conformance of the semantic library is evaluated.

This chapter is structured as follows. First, Section 6.1 describes the evaluation approach of the semantic library and Semantfiyr. Next, Section 6.2 introduces and evaluates the Crossroads example model. Finally, Section 6.3 overviews and evaluates the approach using the more complex Simple Space Mission model.

6.1 Evaluation Approach

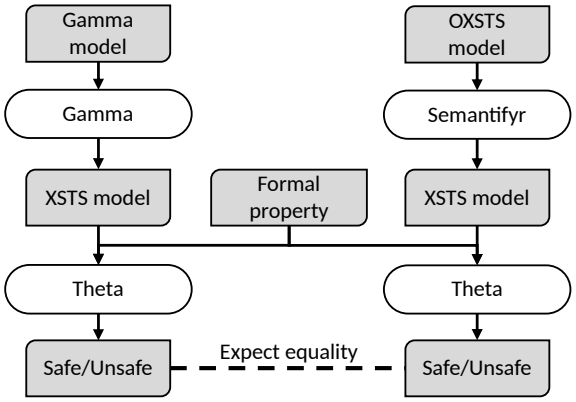


Figure 6.1: The evaluation approach of comparing the OXSTS and Gamma representations of the System.

To validate the conformance of the Gamma Semantic Library to the Gamma semantics, somehow the semantics of the XSTS models must be compared. However, such models are hard to compare, small syntactical differences can result in huge semantical changes. Therefore, the semantics of the models must be compared directly. When the models are considered as black boxes, then for all intents and purposes, two models can be considered equal, *iff* there are no experiments that produce different outputs. Figure 6.1 depicts the approach. By modeling the system in both Gamma and OXSTS, and transforming them to their respective XSTS representations, one can define various *formal properties*.

Such formal properties can be checked by Theta: returning either *Safe* or *Unsafe* result (see Section 2.4.2). By comparing the results of the same properties, we can compare the semantics of the two models. The rest of the chapter models two high-level systems using Gamma and OXSTS as well, and validates the conformance of the Gamma Semantic Library to the Gamma semantics using the above defined approach.

6.2 Crossroads

The Crossroads [20] model is a high-level example from the Gamma repository. It is used as a tutorial introducing the Gamma language and framework, however, it is perfect for demonstrating how to model real systems in OXSTS, as well as for checking the semantical and behavioral conformance of the semantic library.

6.2.1 System

The Crossroads system (see Figure 6.2) 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 yellow: Blank → Yellow → Blank.

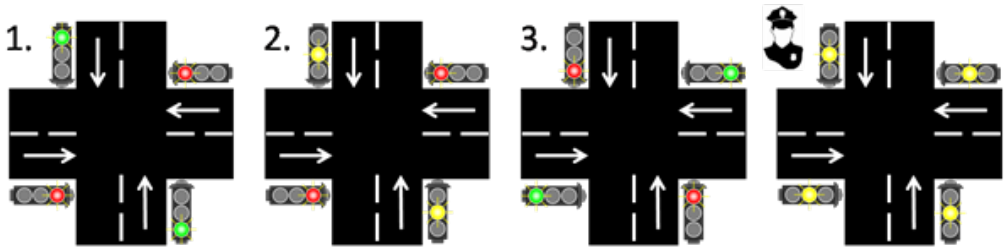


Figure 6.2: The behavior of the Crossroads System from the Gamma Tutorial model [20].

The system is controlled by a central Crossroads Controller component, as depicted in Figure 6.3. 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.

The SysML representation of the TrafficLigthCtrl component is shown in Figure 6.4. 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.

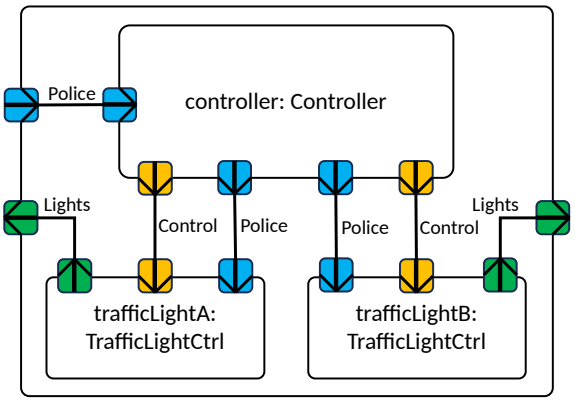


Figure 6.3: The Crossroads Controller component.

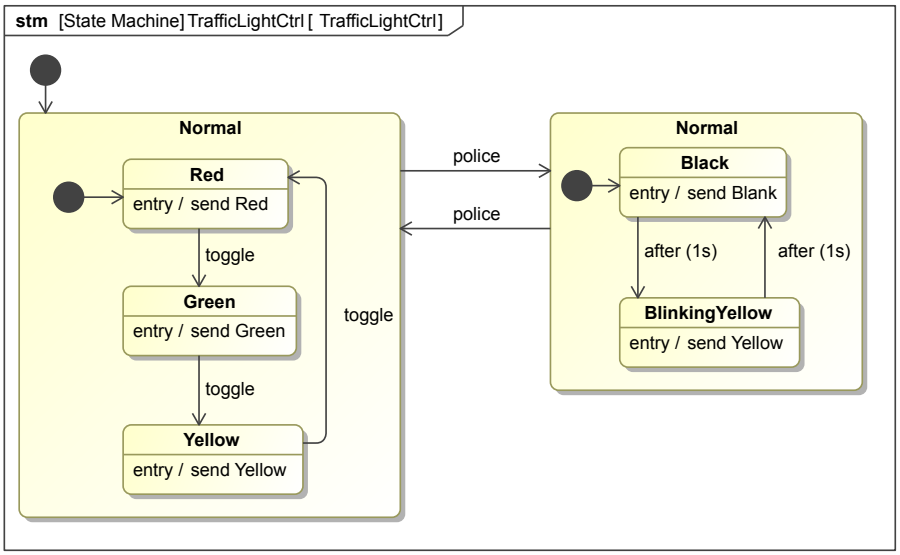


Figure 6.4: The TrafficLightCtrl State Machine model.

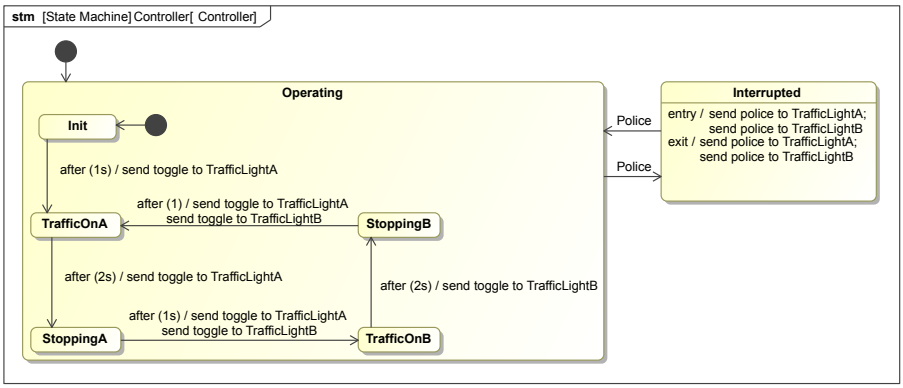


Figure 6.5: The Controller State Machine model.

The SysML model of the Controller component is shown in Figure 6.5. 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 forwards the police event to the two TrafficLight components. Note that the transition between Operating and Interrupted has a higher priority than internal ones.

6.2.2 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 6.1: The interface definitions of the Crossroads system [43].

As a reference, the Gamma version of the Crossroads model is presented [20]. For more information on Gamma, see Section 2.5. First, the interfaces are defined in Listing 6.1. Listing 6.3 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 6.4 shows the Gamma model of the TrafficLightCtrl component. The final Crossroad Controller component is defined in Listing 6.2.

```
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 6.2: The Crossroad Controller component modeled in Gamma [43].

```
statechart Controller [
 port Police : requires PoliceInterrupt
 port ControlA : provides Control // same for B
 port PoliceA : provides PoliceInterrupt // same for B
] {
 timeout TrafficTimeout // timeout specification
 region Main {
  initial Entry0
  state Operating {
   region operating {
    initial Entry1
    state Init { 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 6.3: The Controller component behavior modeled in the Gamma language [43].

```
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 Black to BlinkingYellow when timeout BlinkingTimeout
}
```

Listing 6.4: The TrafficLightCtrl component behavior modeled in Gamma [43].

6.2.3 OXSTS Model

Using the Gamma semantic library modeled in Section 5.1, the Crossroad model can be replicated in OXSTS. An excerpt of the Controller component model is shown in Listing 6.5. The various events are instantiated directly in the type, and placed into the input-output events feature, depending upon their direction. Transitions are modeled in a similar fashion. To listen to events *EventTriggers* are used, while the elapsing of time is modeled using *TimeoutTriggers*. To save space the Operating state is continued in Listing 6.6. The traffic light controller is modeled similarly, of which an excerpt is shown in Listing 6.7. The whole system is constructed in Listing 6.8. The full models are available in the Semantifyr GitHub repository.

```
type Controller : Statechart {
 containment policeEvent :> inputEvents : Event
 containment policeEventA :> outputEvents : Event
 containment toggleEventA :> outputEvents : Event // same for B
 containment trafficTimeout :> timeouts : Timeout
 containment Main :> regions : Region {
  containment et1 :> entryTransitions : EntryTransition { reference ::> to : State = Operating }
  containment Operating :> states : State { containment OperatingRegion :> regions : Region }
  containment operatingToInterrupted :> transitions : Transition {
   reference ::> from : State = Operating
   reference ::> to : State = Interrupted
   containment t :> trigger : EventTrigger { reference ::> event : Event = policeEvent }
  }
  containment interruptedToOperating :> transitions : Transition // ...
  containment Interrupted :> states : State // ...
} }
```

Listing 6.5: Excerpt of the Controller behaviour modeled in OXSTS using the Gamma Semantic library.

```
containment Operating :> states : State {
 containment OperatingRegion :> regions : Region {
  containment et2 :> entryTransitions : EntryTransition { reference ::> to : State = Init }
  containment Init :> states : State {
   containment ea1 :> entryActions : SetTimeoutAction {
    reference ::> timeout : Timeout = trafficTimeout
   }
  }
  containment initToTrafficOnA :> transitions : Transition {
   reference ::> from : State = Init
   reference ::> to : State = TrafficOnA
   containment t :> trigger : TimeoutTrigger {
    reference ::> timeout : Timeout = trafficTimeout
   }
   containment a :> actions : RaiseEventAction { reference ::> event : Event = toggleEventA }
  }
  containment TrafficOnA :> states : State // ...
  containment stoppingBToTrafficOnA :> transitions : Transition // ...
} }
```

Listing 6.6: Excerpt of the Operating state of the Controller statechart modeled in OXSTS.

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

Listing 6.7: Excerpt of the TrafficLightCtrl behaviour modeled in OXSTS using the Gamma Semantic library.

```
target CrossroadsMission {
 containment crossroad : Crossroad
 init { inline crossroad.init() }
 tran {
  inline crossroad. controller .policeEvent.havoc()
  inline crossroad.main()
  inline crossroad.passTime()
} }
```

Listing 6.8: Constructing the CrossroadsMission.

6.2.4 Experimental Evaluation

I evaluated the conformance of the Gamma Semantic Library to the Gamma semantics in two steps: first I validated the semantic conformance, then the specification conformance.

Semantic Conformance

Semantic conformance means the model does not behave inconsistently, such as incorrect state activations. 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 and transitions.

- If a composite state is active, each other composite state 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. An example Target definition is depicted in Listing 6.9. The specified formal requirement asserts that the *Interrupted* state is never active. Theta is expected to return an *unsafe* result with an execution trace signaling that the requirement does not hold, in which case the original formal requirement is proven.

```
1 target ControllerInterrupted_Unsafe : CrossroadsMission {
2  prop { crossroad.controller.Main.activeState != crossroad.controller.Main.Interrupted }
3 }
```

Listing 6.9: The Target definition checking whether the Interrupted state is reachable in the Controller statechart.

Specification Conformance

Specification conformance means that the model behaves as the system specified. Usually, such a test is called the *verification* of the system to the high-level requirements specified by the systems engineers. Since there are no high-level requirements for this model I defined some myself.

- All states must be reachable;
- Both TrafficLightCtrls must be Interrupted at the same time;
- Only allowed pairs of lights must be active, see Table 6.1.

Table 6.1 lists all state-configurations the two traffic lights may be in during Normal operation, annotated with whether it is allowed by the specification or not. Since the two lights follow a strict sequence of states, some state configurations are impossible in a correct model, such as both being green at the same time (which is a gross safety violation).

TrafficLightA	TrafficLightB	Specification	Result	Fault Injection
Red	Red	✓	✓	✓
Red	Green	✓	✓	✓
Red	Yellow	✓	✓	✓
Green	Red	✓	✓	✓
Green	Green	✗	✗	✓
Green	Yellow	✗	✗	✓
Yellow	Red	✓	✓	✓
Yellow	Green	✗	✗	✓
Yellow	Yellow	✗	✗	✓

Table 6.1: All state-configurations the two TrafficLightCtrl components may be in (during Normal operation). Each configuration is annotated with the specifications and the results of the two tests. ✓ means allowed, while ✗ means prohibited.

Conformance Results

A total of 40 formal properties are constructed for the conformance tests. The models, along with an automated conformance test, can be viewed on the Semantifyr GitHub repository. From the 40 requirements, all 40 produced the expected results, both for the Gamma model and the OXSTS model.

Testing With Fault Injection

To further validate the above results, we can use fault injection. A variant of the Controller component was constructed that injects some faults into the model, depicted in Listing 6.10. By removing some actions, the police event is no longer forwarded to the TrafficLightB component, thus the two traffic lights can get out of sync with each other when police mode is activated. The fault injection results are shown in Table 6.1. The results are as expected: resynchronization allows each configuration previously prohibited by the model.

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

```
containment Interrupted :> states : State {
    containment ea1 :> entryActions : RaiseEventAction {
        reference ::> event : Event = policeEventA
    }
// Removed lines:
//     containment ea2 :> entryActions : RaiseEventAction {
//         reference ::> event : Event = policeEventB
//     }
    containment ea3 :> exitActions : RaiseEventAction {
        reference ::> event : Event = policeEventA
    }
// Removed lines:
//     containment ea4 :> exitActions : RaiseEventAction {
//         reference ::> event : Event = policeEventB
//     }
}
```

Listing 6.10: Removed lines in the Interrupted state of the Controller component.

- 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.
- Incorrect light configurations (see Table 6.1).

This evaluation shows that OXSTS and Semantifyr are indeed capable of replicating systems defined in high-level languages. The various conformance checks demonstrated the conformance of the Gamma Semantic Library to the Gamma semantics defined in Section 2.5.

6.3 Simple Space Mission

The Simple Space Mission model [48] is a more complex model created by the OpenMBEE community. The SysML model describes the communication of a satellite with limited battery charge with a ground station, where data transfer costs power. To protect itself, the satellite shuts off communication when under a specific level of charge, and begins to recharge. However, the recharge, data transfer, and power depletion are modeled in a highly asynchronous fashion, using composite states, parallel regions, and channel-based communication.

The state-based behavior of the station can be seen in Figure 6.7. The station is Idle initially, only starting *Operation* when the *Start* event is received or after 30 seconds. When the Operation state is entered, the *Spacecraft* is pinged – asking for data – and the *Receive data* internal behavior is activated. The ping signal is resent after each received data or 10 seconds after the last one arrives.

The behavior of the Spacecraft is shown in Figure 6.6. It has two parallel regions: one responds to ping signals, and the other handles battery recharging. The spacecraft only responds until the battery becomes low (under 30%), at which point it stops transmitting. Recharging begins under 80% and is done until fully charged.

The OXSTS representation of the Simple Space Mission is based on the Gamma model [23], which is again used as a semantic reference point. Note, that the Gamma model is an approximation of the original semantics, it models the internal activity behaviors using parallel regions. The Station component is a *Statechart* with ping and data events, as well as a few timeouts. The data variables are modeled using variables. Internal composite states, parallel regions, and transitions are modeled similarly to previous examples. An excerpt of the type is depicted in Listing 6.11. The spacecraft and station components are connected in the *SpaceMission* component and are put together in the *Mission* Target definition, an excerpt of which is shown in Listing 6.12. The Target first havocs the two environment events, runs the component, and then passes the time.

The requirements for the system are constructed using the following approach.

- Each state must be reachable – 9 cases.
- The satellite must send all of the data – 11 cases.
- The station must receive all of the data – 11 cases.
- The charge of the satellite must stay within the $20\% - 100\%$ range – 9 cases.

Listing 6.12 also contains two example test verifications testing that the battery charge never falls under 20%, and that 100% of the data is received by the station. Of the 40 verification cases defined each verification case was checked, except the most complex ones. The verification cases that entail running the whole scenario proved too difficult for Theta. The reason is that Theta does not behave well with asynchronous timed models, because it can not represent them efficiently in its abstract states. However, checking these cases is possible by using another integrated model checker of Gamma, such as UPPAAL [39], which can verify timed systems efficiently. Another solution is integrating a new approach of using lazy-abstraction over explicit clock variables presented in [15].

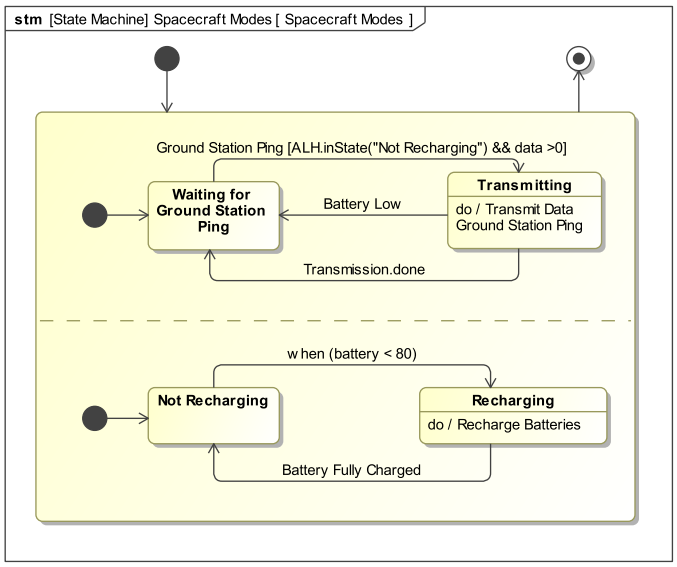


Figure 6.6: The State Machine behavior of the Spacecraft component.

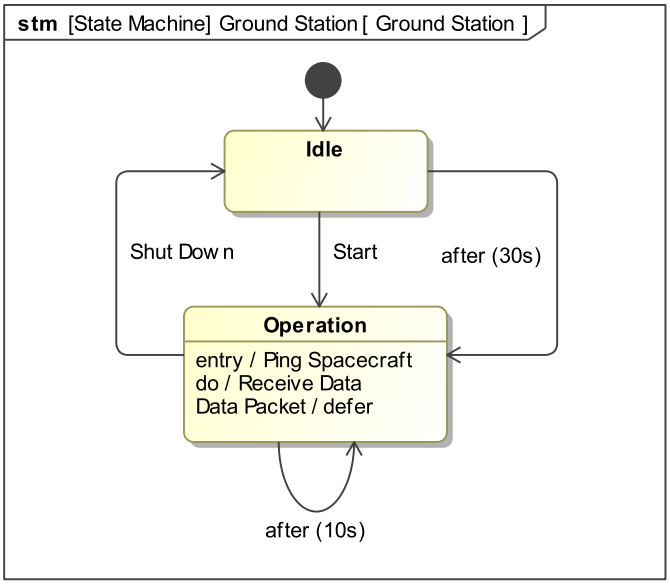


Figure 6.7: The State Machine behavior of the Station component.

```
type Station : Statechart {
 containment dataEvent :> inputEvents : Event
 containment startEvent :> inputEvents : Event
 containment shutdownEvent :> inputEvents : Event
 containment pingEvent :> outputEvents : Event
 containment pingTimeout :> timeouts : Timeout
 containment startTimeout :> timeouts : Timeout
 var receivedData : Integer = 0
 containment Main :> regions : Region {
  containment et1 :> entryTransitions : EntryTransition { reference ::> to : State = Idle }
  containment Idle :> states : State {
   containment ea1 :> entryActions : SetTimeoutAction {
    reference ::> timeout : Timeout = startTimeout
    reference ::> amount : Integer = 1500
   }
  }
  containment startOperation :> transitions : Transition {
   reference ::> from : State = Idle
   reference ::> to : State = Operation
   containment t :> trigger : EventTrigger { reference ::> event : Event = startEvent }
  }
  containment Operation :> states : State // ...
 }
}
```

Listing 6.11: Excerpt from the Station component.

```
type SpaceMission : SyncComponent {
    containment station :> components : Station
    containment spacecraft :> components : Spacecraft
    containment pingChannel :> channels : Channel {
        reference ::> inputEvent : Event = station.pingEvent
        reference ::> outputEvent : Event = spacecraft.pingEvent
    }
    containment dataChannel :> channels : Channel {
        reference ::> inputEvent : Event = spacecraft.dataEvent
        reference ::> outputEvent : Event = station.dataEvent
    }
}
target Mission {
    containment spaceMission : SpaceMission
    init { inline spaceMission.init() }
    tran {
        inline spaceMission.station.startEvent.havoc()
        inline spaceMission.station.shutdownEvent.havoc()
        inline spaceMission.main()
        inline spaceMission.passTime()
    }
}
target Spacecraft_batteryCharge_never_under_20_Safe : Mission {
    prop { ! (spaceMission.spacecraft.batteryCharge <= 20) }
}
target Station_ReceivedData_100_Unsafe_Slow : Mission {
    prop { ! (spaceMission.station.receivedData == 100) }
}
```

Listing 6.12: The SpaceMission component, the Mission target definition, as well as an example verification case.

Chapter 7

Conclusion and Future Work

In this work, I addressed challenges associated with employing formal methods in model-based systems engineering, specifically when dealing with ontology-based modeling languages. By combining ontological modeling with operational semantics, I proposed an approach that extends ontological engineering languages with operational semantic libraries. This methodology leverages the structural benefits of ontological languages while utilizing operational semantics for temporal aspects, thereby enhancing the feasibility of formal verification.

To demonstrate this approach, I introduced OXSTS, a new ontological-operational language, and developed Semantifyr, a tool capable of interpreting it. Using Semantifyr, I modeled the semantics of Gamma in a semantic library and evaluated it using case studies of high-level SysML models. The evaluation showed that Semantifyr can indeed represent high-level languages and their models, allowing the direct operationalizing of such languages. The case studies also validated the initial implementation of the Gamma Semantic Library, by confirming the behavioral conformance of the formal models to their original SysML and Gamma counterparts. The results indicate that my proposed approach is viable, and deserves further study.

Future work will focus on implementing the methodology within existing ontological modeling languages, particularly KerML, which shows potential for defining high-level domain-specific languages, such as SysML v2 and AADL. By solving the instantiation problem and incorporating operational elements into KerML, the formal verification of derived languages could be realized. As direct next step, I plan to implement the KerML/SysML v2 semantic libraries in OXSTS to demonstrate the power of Semantifyr and support the formal verification of SysML v2 models. Additionally, to improve the usability of OXSTS, I plan to extend the current IDE support with model debugging, as that seemed to be the biggest hurdle during modeling. Finally, I will integrate Semantifyr into the Gamma framework, leveraging its model-slicing optimization and model checker integration features.

Chapter 8

Acknowledgement

I would like to express my gratitude to my advisor, Vince Molnár, who has continuously provided me with guidance, valuable ideas, and feedback during this work. I would also like to thank my former advisors, András Vörös, Bence Graics, and Márton Elekes, for their continued support during the past six years of university. This work could not have been possible without them!

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

Chapter 9

Köszönetnyílvánítás

Számomra ez a diplomaterv a tökéletes lezárása az elmúlt időszaknak, egy mérföldkő az életemben. Ezért szeretném megragadni az alkalmat, hogy itt is köszönetet mondjak mindazoknak, akik támogatásukkal segítettek az elmúlt hat év során.

Egyetemi éveim alatt számos kihívással találkoztam, amelyek formáltak engem. Ezen utazás nem lett volna lehetséges a rengeteg támogatás nélkül, amit kaptam. Először is, köszönöm *családomnak*, hogy mindenben támogattak, és akkor is megértőek maradtak, amikor nem tudtam elég időt szánni rájuk. *Deli*, *Kristóf*, *Peti*, *Tina*, *Zsófi* —- nélkületek nem sikerült volna!

Köszönöm a *Kritikus Rendszerek Kutatócsoport* minden tagjának, különösen *Vörös Andrásnak*, hogy elsőévesként felkarolt és elindított ezen az úton, *Elekes Mártonnak* és *Graics Bencének*, a TDK dolgozataim konzultálásáért és szakadatlan támogatásukért, *Micskei Zoltánnak* pályámban való rengeteg segítségéért, valamint *Semeráth Oszkárnak* és *Marussy Kristófnak* a megannyi éjszakába nyúló diskurzusokért.

Különösen köszönöm konzulensemnek, *Molnár Vincének*, az elmúlt évek során nyújtott megannyi támogatásáért. Vince, a témavezetői kötelességein túlmenően, szakmailag és személyesen is rengeteget segített értékes beszélgetésekkel és tanácsokkal. Témavezetése alatt nemcsak szakmailag, hanem *emberileg* lettem több, ezért hálás köszönet illeti őt.

Köszönöm szakmai vezetőimnek, *Hegedűs Ábelnek* és *Debreceni Csabának* a rengeteg támogatást és tanácsot az elmúlt évek során. Bennetek bármikor megbízhattam, és mindig őszinte maradhattam. Rengeteget tanultam tőletek, nem csak szakmáról, hanem az életről egyaránt.

Köszönöm *barátaimnak*, akik a kezdetektől jobb emberré formáltak, és akikkel együtt jártuk végig e göröngyös utat: *Bacus*, *Balázs*, *Csó*, *Fruzsi*, *Gábor*, *Lili*, *Martin*, *Máté*, *Viktor* – veletek minden sokkal egyszerűbb volt.

És végül, de nem utolsó sorban, köszönöm Neked, kedves olvasó, hogy időt szántál arra, hogy elolvasd életem egyetemi évének eredményét. Remélem hasznosnak találtad, de ha nem is, legalább értékelted a szép ábrákat, bonyolult mondatokat, és a színes kódsorokat.

Most pedig kezdődjön életem történetének következő fejezete!

Bibliography

[1] Islam Abdelhalim, Steve Schneider, and Helen Treharne. An Optimization Approach for Effective Formalized fUML Model Checking. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, pages 248–262, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33826-7.

[2] 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.

[3] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 10 2005. DOI: 10.1016/j.artint.2005.05.003.

[4] 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.

[5] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In Maria Carla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 236–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45798-5.

[6] 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.

[7] Conrad Bock and James Odell. Ontological Behavior Modeling. *Journal of Object Technology*, 10:3: 1–36, 01 2011. DOI: 10.5381/jot.2011.10.1.a3.

[8] Alexander Borgida and Ronald Brachman. Conceptual Modeling with Description Logics. In *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 349–372, 01 2003.

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

[10] Ruirui Chen, Yusheng Liu, and Xiaoping Ye. Ontology Based Behavior Verification for Complex Systems. In *International Design Engineering Technical Conferences*

and Computers and Information in Engineering Conference, page V01BT02A038, 08 2018. DOI: 10.1115/DETC2018-85689.

[11] Ajay Chhokra, Sherif Abdelwahed, Abhishek Dubey, Sandeep Neema, and Gabor Karsai. From system modeling to formal verification. In *2015 Electronic System Level Synthesis Conference (ESLsyn)*, pages 41–46, 2015.

[12] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi, and Tullio Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Automated Software Engineering*, pages 362–365. ACM, 2012.

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

[14] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[15] Dóra Cziborová and Richárd Szabó. Modeling of Time-Dependent Behavior in Fault-Tolerant Systems. In *Proceedings of the 31th Minisymposium*, pages 55–60, 2024. DOI: 10.3311/MINISY2024-011.

[16] Philippe Dhaussy, Jean-Charles Roger, and Frederic Boniol. Reducing State Explosion with Context Modeling for Model-Checking. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, pages 130–137, 2011. DOI: 10.1109/HASE.2011.24.

[17] Maged Elaasar, Nicolas Rouquette, Klaus Havelund, Martin Feather, Saptarshi Bandyopadhyay, and Alberto Candela. Autonomica: Ontological Modeling and Analysis of Autonomous Behavior. *INCOSE International Symposium*, 33(1):1570–1585, 2023. DOI: 10.1002/iis2.13099.

[18] 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. DOI: 10.1007/s11219-023-09617-5.

[19] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879.

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

[21] Mario Gleirscher and Diego Marmsoler. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering*, 25(6):4473–4546, Nov 2020. DOI: 10.1007/s10664-020-09836-5.

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

[23] 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. DOI: 10.1007/s10270-020-00806-5.

[24] Bence Graics, Milán Mondok, Vince Molnár, and István Majzik. Model-based testing of asynchronously communicating distributed controllers. In Javier Cámara and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software*, pages 23–44, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-52183-6.

[25] Henson Graves and Yvonne Bijan. Using formal methods with SysML in aerospace design and engineering. *Annals of Mathematics and Artificial Intelligence*, 63(1): 53–102, Sep 2011. DOI: 10.1007/s10472-011-9267-5.

[26] Jan Friso Groote and Frits W. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *International Colloquium on Automata, Languages and Programming*, 1990. URL https://api.semanticscholar.org/CorpusID:267813466.

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

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

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

[30] Nicola Guarino, Daniel Oberle, and Steffen Staab. *What Is an Ontology?*, pages 1–17. Handbook on Ontologies, 05 2009. DOI: 10.1007/978-3-540-92673-3_0.

[31] 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.

[32] 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.

[33] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. DOI: 10.1016/0167-6423(87)90035-9.

[34] 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.

[35] 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*, 2023. DOI: 10.1002/sys.21679.

[36] John C. Knight. Challenges in the utilization of formal methods. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 1–17, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49792-9.

[37] Martin Kölbl, Stefan Leue, and Hargurbir Singh. From SysML to Model Checkers via Model Transformation. In *Proc. of the 25th International Symposium on Model Checking Software*, volume 10869 of *Lecture Notes in Computer Science*, pages 255–274. Springer, 2018.

[38] Saul A. Kripke. Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963. DOI: 10.1002/malq.19630090502.

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

[40] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Comput. Surv.*, 52(6), oct 2019. DOI: 10.1145/3354584.

[41] 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. DOI: 10.1007/s10270-022-01056-3.

[42] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, Oct 2011. DOI: 10.1007/s10270-010-0157-9.

[43] 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.

[44] D. Nardi and R. J. Brachman. *An Introduction to Description Logics*, page 1–44. Cambridge University Press, 2007.

[45] Faranak Nejati, Abdul Azim Abdul Ghani, Keng-Yap Ng, and Azmi Jaafar. Handling state space explosion in verification of component-based systems: A review. *CoRR*, abs/1709.10379, 2017. URL http://arxiv.org/abs/1709.10379.

[46] Object Management Group. *Kernel Modeling Language (KerML)*, 2023. ptc/23-06-01.

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

[48] OpenMBEE. OpenMBEE Simple Space Mission. https://github.com/Open-MBEE/OMGSpecifications, 2019. Accessed: 2024-03-29.

[49] Radek Pelánek. Fighting State Space Explosion: Review and Evaluation. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, pages 37–52, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03240-0.

[50] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, sep 1977. DOI: 10.1145/356698.356702.

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

[52] Charlotte Seidner and Olivier H. Roux. Formal Methods for Systems Engineering Behavior Models. *IEEE Transactions on Industrial Informatics*, 4(4):280–291, 2008. DOI: 10.1109/TII.2008.2008998.

[53] 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.

[54] 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.

[55] Antti Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21.

[56] Emile van Gerwen, Leonardo Barbini, and Thomas Nägele. Integrating System Failure Diagnostics Into Model-based System Engineering. *INSIGHT*, 25(4):51–57, 2022. DOI: 10.1002/inst.12412.

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

[58] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. The hidden models of model checking. *Software and Systems Modeling*, 11(4):541–555, 2012.

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

[60] Lan Yang, Kathryn Cormican, and Ming Yu. Ontology-based systems engineering: A state-of-the-art review. *Computers in Industry*, 111:148–171, 2019. DOI: 10.1016/j.compind.2019.05.003. URL https://www.sciencedirect.com/science/article/pii/S0166361518307887.

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