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

# Abstraction-based Trace Generation to Validate Semantics of Formal Verifiers

**Scientific Students' Association Report**

Author:

Zsófia Ádám

Advisor:

Zoltán Micskei, PhD

2022

# Contents

# Kivonat

A biztonságkritikus beágyazott rendszerek komplexitása folyamatosan növekszik. Ennek a komplexitásnak a kezelésére többek között különböző mérnöki modellek (pl. állapotgépek) tervezésben és verifikációban való használata ad megoldást. Formális verifikációs technikák segítségével bizonyos tulajdonságok teljesülése bizonyítható vagy hibák is megtalálhatóak, de ehhez a mérnöki modellezési nyelv pontos végrehajtási szemantikájának definiálására van szükség.

A formális verifikációs eszközökben szükség van ezen szemantikák formalizálására a bemeneti modell formális reprezentációra transzformálása során. Ez egy komplex elméleti és implementációs feladat, mivel a szemantika sokszor fél-formális és alulspecifikált. Ennek köszönhetően bizonyos hibák, például a koncepció félreértése (mely gyakran az alulspecifikált részek kiegészítésére adott ad-hoc döntésekben nyilvánul meg), gyakoriak. Ezen problémák később a felhasználó zavarodottságát vagy az eszköz által adott eredmények észrevétlen érvénytelenségét is eredményezhetik.

Ennélfogva a szemantikát implementáló modell-transzformáció validálása egy nélkülözhetetlen lépés. A jelenlegi gyakorlat teszt modellek készítése az adott modellezési nyelven majd érvényes lefutások megadása (általában manuálisan). Ezeket ezután a verifikációs eszköz vagy szimulátor által visszaadott lefutásokkal hasonlítják össze a konformancia bizonyítására. Ez a módszer hibákra hajlamos és nem hatékony, mivel bizonyos érvényes lefutások könnyen kihagyásra kerülhetnek.

A teszt lefutások automatikus generálása egy aktív kutatási terület. Minden lehetséges lefutást legenerálni egyszerű modellekre és modellezési nyelveken könnyű feladat. Azonban a feladat nehézsége könnyen megnövekedhet: például ha a modell nem csak vezérlési folyamot ír le, de adatokat is tartalmaz vagy ha az állapottere végtelen. Ilyen esetekben a lefutásokat legtöbbször valamilyen fedettségi kritérium teljesítésére korlátozzák. Azonban a probléma specifikus eseteiben precízebb megoldások is adhatóak.

A formális verifikációs eszközök egy gyakori típusa az absztrakció-alapú model checker. Munkámban ezen eszközök absztrakció alapú technikáit kihasználó automatikus lefutás generálási módszert javaslok. Egy olyan absztrakció alapú lefutás generáló algoritmust alkottam meg, mely sokszor képes a végtelen állapotterek kezelésére. Az absztrakciót konfigurálható módon képes alkalmazni és nem generál olyan lefutásokat, melyek szükségtelenül ismétlik a modell már korábban lefedett állapotait. A dolgozatomban megvizsgálom az algoritmus garanciáit, erősségeit és gyengeségeit, beleértve különböző elérhető fedettségeket és a skálázhatóságot.

Az algoritmust egy esettanulmány során ki is értékeltem reaktív állapotgépekhez készült eszközökön. A validált eszközök saját modellezési nyelvet alkalmaznak, mely megfelelően összetett ahhoz, hogy validálása szükséges legyen. A bemutatott elsődleges felhasználás mellett ismertetem, hogy az algoritmus milyen más felhasználásokra ad lehetőséget, mint például modellezési hibák azonosítása.

# Abstract

Safety-critical embedded systems are becoming increasingly complex. To handle this complexity, various engineering models are used for design and verification during development (e.g. state machines). Formal verification techniques can automatically prove some properties or find errors in these models, but necessitate the precise definition of execution semantics of the engineering modeling language.

Formal verification tools need to formalize semantics when transforming the engineering model to a formal representation. This is a complex theoretical and implementation task, as these semantics are usually only semi-formal and underspecified. Thus different errors, e.g. conceptual flaws, such as unexpected decisions on underspecified semantic rules, are common. Such undetected flaws cause either confusion or silently invalidate the results of formal verification.

Therefore the validation of model transformations implementing semantics is a crucial task. The current state of practice recommends creating test models in the engineering language, defining valid execution traces for them (mostly manually), and later comparing these traces to the ones returned by verifiers or simulators to check conformance. This is an inefficient and error-prone method as valid traces can be easily missed.

Generating test traces from models automatically is an active research area. All possible traces can be easily generated for simple models and modeling languages. However, it gets increasingly difficult when the modeling language contains data and not just control flow, or the model's state space is infinite. Limiting traces to satisfy some kind of coverage criteria is the general solution, but for specific scenarios, more precise solutions can be recommended.

A common type of formal verification tools is abstraction-based model checkers. I propose utilizing the abstraction-based techniques of these tools for automatic trace generation. I designed an abstraction-based trace generation algorithm, which is able to handle several cases where the state space is infinite. It utilizes the abstraction in a configurable way and generates traces that do not unnecessarily repeat already covered states of the model. In this report, I consider the guarantees, strengths, and weaknesses of this algorithm, such as coverage possibilities and scalability.

I evaluated the algorithm through a case study on tools for reactive state machines. The validated tools include their own modeling language, which is complex enough that the need for validation of the transformation arises. Besides this presented main use case, I show that this algorithm also opens up the possibility for more features, such as mitigating modeling mistakes.

# Chapter 1

# Introduction

With the increasing complexity of *safety critical systems*, both *verification techniques* and *model driven development* are experiencing a growing importance. Models play a central role in the design and development of these systems and thus the verification of design models is a crucial step [42, 47]. There are several complementary verification techniques available, such as testing, simulation or *formal verification*.

*Formal verification techniques* offer the capability of not only being able to find *(error) property violations*, but also the absence of them [27], proving the system safe for the given property. One of the best known formal verification techniques is *model checking* [5, 39].

*Model checking* [26] utilizes exhaustive state space traversal. But naive state space traversal is computationally expensive and often made infeasible due to *state space explosion* (i.e. the exponentially growing number of possible states). Many techniques were proposed and are in use to mitigate state space explosions, such as bounded model checking [18], symbolic methods [20] or *abstraction-based techniques* [25, 26].

There are many different *model checking tools* in different application domains [43, 10], implementing these algorithms and enabling their users to automatically check different models (e.g. software code [10], hardware models [43] and so on). But these tools implement much more than a single algorithm: they execute complex processes including the transformation of the input model to an unambiguous formal representation, optimization on this formal model and the backannotation of the results to the original model [6].

*Formal representations* are necessary for an algorithm to reason upon the model with mathematical precision. However there is a large semantic gap between engineering design models and formal models, as design models tend to be semi-formal and ambiguous. This makes the mapping between the two a non-trivial and complex task. Although model checking algorithms are typically proven to be correct [25], this is often not the case for the *model transformation* and optimization steps preceding the model checking algorithm.

**Problem Statement** Due to this semantic gap and the complexity of the verification process, subtle semantic and implementation issues might be introduced in the *model transformation* and optimization steps and these issues can easily remain hidden. If the formal model is syntactically correct, but *semantically inaccurate*, i.e. it has different behaviour from the original model, then the results of the model checker should be invalidated.

For example the tool might not find issues that are present in the input model, but absent from the formal counterpart causing a missed bug, which will most likely go unnoticed. So the question arises: when can we trust the results of a formal verification tool?

**Solution Proposal** I propose the *end-to-end (E2E) validation* of verification processes to find issues in the semantics implemented by the model transformation process.

Engineering modeling language semantics are typically *not fully formalized* and therefore the validation process needs to include manual checks. An intuitive and typical method for validation of model semantics is to check what executions the model is capable of through execution traces of a conformance test suite [53].

I propose the *automatic generation of execution traces* using the model checker under validation itself. Although the validation approach is partially manual, it can be assisted by automated tools. The scope of the algorithm proposed in this work are mainly abstraction based model checkers.

Generating test cases through counterexample traces of model checkers is an already widely used approach [33], but it typically suffers from issues mainly caused by the blackbox usage of model checkers [32].

The proposed novel algorithm utilizes lower level features of model checkers. The *abstraction and state space traversal* capabilities can be utilized for generating execution trace sets with *unique coverage guarantees* for states, transitions and data variables as well.

During trace generation, the *model transformations* and optimizations typically executed during verification are also used. Thus the resulting trace set will reflect the *semantic mapping* applied by the model checker. This makes the generated execution traces appropriate for *E2E validation*.

**Evaluation** To evaluate the algorithm and the validation approach, I created a *prototype implementation* to validate the verification process of Gamma [50] and Theta [58], a toolchain for state machine based reactive systems.

The case study includes the design of a validation model suite, quantitative analysis of the resulting traces and detailed examination of the findings of the validation process. The models and traces are available as an artifact [1]. Based on the results, the approach was deemed successful, as the compact trace set made manual validation feasible and multiple issues and limitations were found in the toolchain.

Furthermore, trace generation was also executed on several real-world models as another case study. This illustrates another use case of trace generation: assisting the understanding of semantics on the concrete model itself and possibly detecting modeling mistakes, i.e. human error. Though it has some limitations, this approach also proved to be feasible.

**Contributions**

- I propose an *E2E validation approach* for semantics of model transformations in abstraction based model checkers (Chapter 3),
- I propose, formalize and analyze the coverage guarantees and usability of a novel *trace generation algorithm* utilizing the state space traversal and abstraction capabilities of model checking tools (Chapter 4),
- I created a *prototype implementation* of the algorithm and systematically design a *validation model suite* (Chapter 5),
- I provide the evaluation of the validation approach through *two case studies*: 1) a case study of the *validation process*, 2) a case study examining trace generation on *real-world models* (Chapter 5).

# Chapter 2

# Background

This chapter provides the necessary background and context for this report: first, the role of formal verification in developing safety critical systems is described in Section 2.1. Then formal verification and model checking is introduced with focus on abstraction-based techniques (Section 2.2). Lastly, the different models throughout model checking and formal verification processes are explained (Section 2.3).

## 2.1 Verification of Critical Embedded Systems

With the rising number of application domains and growing complexity of critical embedded systems, design processes are also becoming more and more complex. A common way to handle complexity is to utilize techniques of model-based systems engineering by creating different design models throughout the whole process. These design models capture the structure and behaviour of the system under development.

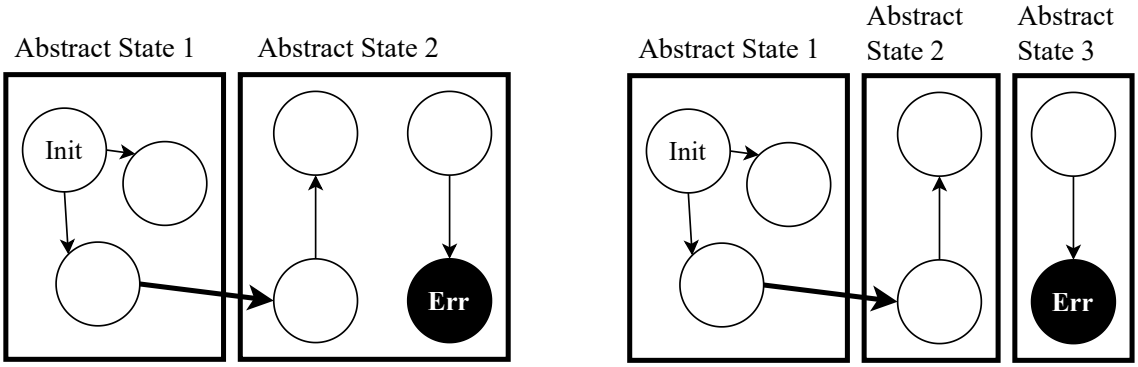
When models are extensively used during design, verification of such engineering models is a crucial part of these processes. Different project goals and types of models require different techniques, such as testing [36] or simulation. These are often complementary techniques used with different goals in mind, as each has their own set of strengths and weaknesses. Some typical examples are as follows.

**Simulation** is capable of executing the models, usually on the main scenarios to let the user examine the behaviour of the model through execution traces. Simulation is widely used in practice to check software, hardware, mechanical etc. design.

**Model-Based Testing** in general test cases are often created manually, following a given test design approach. However when utilizing model-based testing, models can be used for automatic test case generation following pre-defined coverage criteria. Extensive research work and many different methods exist in this topic [19, 40].

**Conformance testing** is a specific problem in testing, which aims to uncover whether the model and the implementation of the system has the same observable behaviour. Conformance test suites can be created manually [41], or generated based on models (e.g. with the W or Wp method [19]).

**Formal Verification** stands for methods that are capable of automatic reasoning upon a formal model to prove violation or correctness regarding a given property.




**(a)** Improper abstraction level causing a false positive result.

**(b)** Proper abstraction level proving *Err* unreachable.

**Figure 2.1:** Model checking with abstraction: the circles are concrete states of the model, while the rectangles are abstract states. The goal is to prove reachability of the *Err* state.

Testing and simulation is a typical technique for many systems. But when a system is critical enough to require additional proofs of correctness, formal verification techniques also have to be utilized.

## 2.2 Formal Verification and Model Checking

Formal verification techniques utilize mathematically precise reasoning over the formal model of a system to prove the violation or satisfaction of given properties.

Model checking [27] is a formal verification technique utilizing automated and exhaustive state space traversals to give counterexamples or proofs of correctness regarding different properties, such as reachability of a given state, termination, variable overflows and so on.

State space traversal, in general, cannot be done efficiently due to the issue of state space explosion: the state space can easily grow exponentially with the number of variables, i.e. a single 32 bit integer can represent $2^{32}$ values, adding a multiplier of $2^{32}$ to the number of possible states.

Tackling state space explosion is one of the main problems of model checking algorithms. There are many well-known techniques, e.g. bounded model checking [18], symbolic methods [20] or abstraction [25, 26].

### 2.2.1 Abstraction in Model Checking

Various abstraction techniques found a common use in many different model checking algorithms. Abstract states can cover several, if not an infinite amount of concrete states. With the right abstraction level the abstract state space becomes small enough for exhaustive traversal, while also proving a violation or correctness, as illustrated in Figure 2.1.

Finding the right abstraction level requires further techniques to be used, e.g. Counterexample-guided Abstraction Refinement (CEGAR) [25], where abstraction and refinement are combined in a loop alternating the two.

#### 2.2.1.1 Abstract Domains

Implementing abstraction requires an *abstract domain*, a *precision* and a *transfer function* to be defined. Informally an abstract domain defines the domain of abstract states, the current precision shows the level of abstraction, while the transfer function defines how the successors of abstract states.

Formally they can be expressed the following way:

**Definition 1 (Abstract Domain [14]).** An abstract domain is a tuple $D = (S, \top, \bot, \sqsubseteq, \mathsf{expr})$ where

- $S$ is a (possibly infinite) lattice of abstract states,
- $\top \in S$ is the top element,
- $\bot \in S$ is the bottom element,
- $\sqsubseteq \subseteq S \times S$ is a partial order conforming to the lattice and
- $\mathsf{expr}\colon S \mapsto \mathit{FOL}$ is the expression function that maps an abstract state to its meaning (the concrete data states it represents) using a first order logic (FOL) formula. ▪

**Definition 2 (Transfer Function [14]).** Let $\pi$ be the precision defining the current precision of the abstraction.

Then the transfer function is $T\colon S \times \mathit{Ops} \times \Pi \mapsto 2^S$, calculating the successors of an abstract state with respect to an operation and $\pi$. ▪

There are many possible abstract domains, e.g. Cartesian predicate abstraction [37], boolean predicate abstraction [7], explicit-value abstraction [12] or even combinations of these and others [4]. However, this work will focus on the explicit-value domain.

**Explicit-value Abstraction [12]** The explicit-value domain introduces a fairly simple method of abstraction, which tries to directly remedy state space explosions by only tracking the value of a subset of the variables.

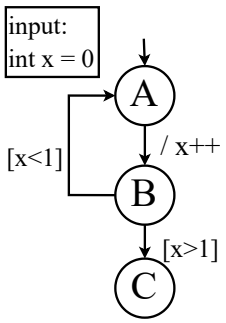
Thus the precision is defined by adding which variables should be tracked and the abstract states contain value assignments to all of the variables, which are made abstract by the capability to assign the value "unknown" ($\top$) to untracked or unassigned variables.

#### 2.2.1.2 ARG

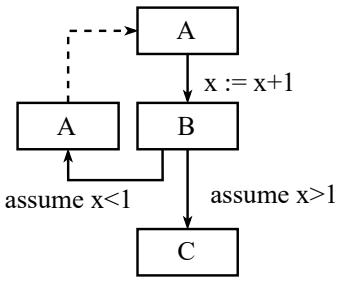
Abstraction based model checkers traverse an abstract state space building an *Abstract Reachability Graph* (ARG) [13].

**Definition 3 (Abstract reachability graph).** An Abstract Reachability Graph is a tuple $\mathit{ARG} = (N, E, C)$ where

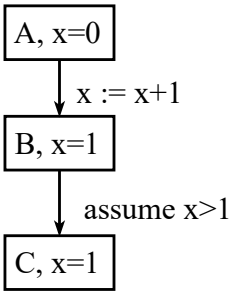
- $N \subseteq S$ is the set of *nodes*, each corresponding to an abstract state in some domain.
- $E \subseteq N \times N$ is the set of directed *edges*. An edge $(s_1, s_2) \in E$ is present $s_2$ is a successor of $s_1$ with regards to the transfer function $T$.




**(a)** An extended finite-state machine (EFSM) with a single variable.




**(b)** The ARG for the EFSM, where $x$ is excluded from the precision. The dashed edge is a covered-by edge.




**(c)** Concretized trace from the ARG.

**Figure 2.2:** Example of an ARG and concretized trace in the explicit domain.

- $C \subseteq N \times N$ is the set of *covered-by edges*. A covered-by edge $(s_1, s_2) \in C$ is present if $s_1 \sqsubseteq s_2$. ▪

The model checker builds the ARG by expanding already existing nodes with their successors and adding *directed edges*. A *covered-by* edge will be used where possible instead of expanding the node. This is done on a given abstraction level, which allows the ARG to stay finite in many cases, even when the concrete state space would be infinite.

#### 2.2.1.3 Traces

Paths inbetween the ARG nodes on the directed edges are called *abstract traces*. If the path leads from the initial node to an erroneus state then the abstract trace is an *abstract counterexample*. If it is feasible, then it can be concretized into a *concrete counterexample*. Of course the feasibility check and concretization can be done on any given abstract trace.

Abstract and concrete traces are illustrated in the example of Figure 2.2. The abstract states (rectangles) of the ARG only include the states of the EFSM, but not the value of $x$. There is an abstract trace in the ARG to C, which can be concretized to the trace shown in Figure 2.2c, where $x$ is now included and thus the states are not abstract anymore. On the other hand, there is another abstract trace $A - B - A$ in the ARG as well, which is infeasible and thus can not be concretized.

## 2.3 Using Model Checkers in Practice

This section intends to give insight on how model checking looks in practice through the typical types of models and modeling languages that can be utilized throughout verification processes. These model types and some examples are listed in Figure 2.3 and explained in the paragraphs below.

**Input Models** Model checking is a widely used method with many different application domains, e.g. hardware specifications [55], software [10], protocols [23, 28], engineering and business models [43, 50]. Thus input models are often design models or software code instead of unambiguous formal models.

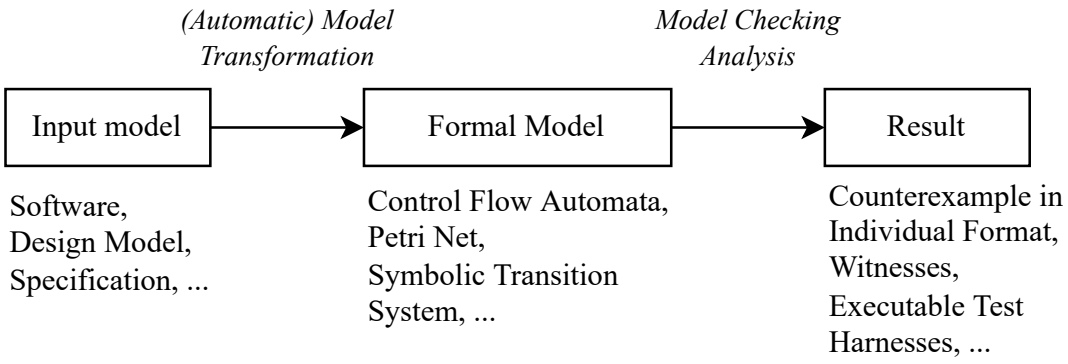



**Figure 2.3:** Typical models throughout the verification process.

In this report the evaluation will focus on state machines modeling reactive systems and thus most examples will also be added as state machines, although the proposed algorithms and processes are not limited to these kind of models.

**Formal Representation** The system under verification has to be an unambiguous formal model as this enables reasoning with mathematical precision over it. It is not unheard of to directly create formal models or manually transform design models or protocol specifications to formal models (e.g. manually creating Petri nets or Extended Finite State Machines).

However, most tools implement an automatic model transformation step instead, which generates a formal representation out of the input design model, for example transforming software code to Control Flow Automata (CFA) [6, 11].

**Results, Counterexamples** Beside a binary result of correct or faulty, model checking tools may also provide a counterexample or a proof of correctness. Counterexamples are concrete traces, usually backannotated to the input model from the formal representation, so they are readable for the user. In some cases, mainly in software model checking, the tool might even be able to generate an executable test harness [17], which runs the faulty execution.

There are also initiatives in software model checking for a uniform format, called witness [15, 16]. This uniformity enables, for example, the validation of the proof or counterexample by a separate verifier.

As described above, formal verification processes are more complex than just their core algorithms and can include many different models. These models and the transformations inbetween all have to be correct in order for the tool's results to be reliable. Making sure of this correctness is a complex question and one of the key motivations of this report. Thus it is further elaborated on in Chapter 3.

# Chapter 3

# Validating Semantics of Verifiers

This chapter introduces the common formal verification process of model checkers in detail (Section 3.1). It describes how issues in model transformation endanger the validity of the verification results (Section 3.2).

Section 3.3 explains why these model transformations are also error prone due to ambiguous semantics, especially if the input model is some kind of engineering model.

Section 3.4 proposes a solution: a validation process based on trace generation, which will serve as the basis for the rest of this report.

## 3.1 Formal Verification Process

The scope of this work will mainly revolve around the formal verification of engineering models with model checking tools. Engineering models are used not just for mutual understanding, but for more and more refined design as well. Due to their growing function importance, formal verification of these models is becoming crucial as well. One of the best-known formal verification approach is model checking.

The typical high-level process implemented for verification in tools or toolchains is shown in Figure 3.1. Although the actual reasoning upon the model is executed in the model checking analysis step, the verification process itself consists of much more steps than that.

**Design Tool** Usually there is a design tool at the beginning of the toolchain, used to create the engineering models (e.g. activity diagrams, state machines, hardware descriptions).

**Engineering Model and Modeling Language** The engineering modeling language might be text-based or visual, but it is certainly operating on a fairly high abstraction level to help usability.

**Model Transformation** Such a model cannot be reasoned upon directly by the model checker, so it goes through a series of model transformation and optimization steps and is transformed into a formal model, which has an unambiguous, formal language that the model checker can work with.

**Model Checking Analysis** The model checking algorithms of the tools are executed on the formal model, as described in Section 2.2.

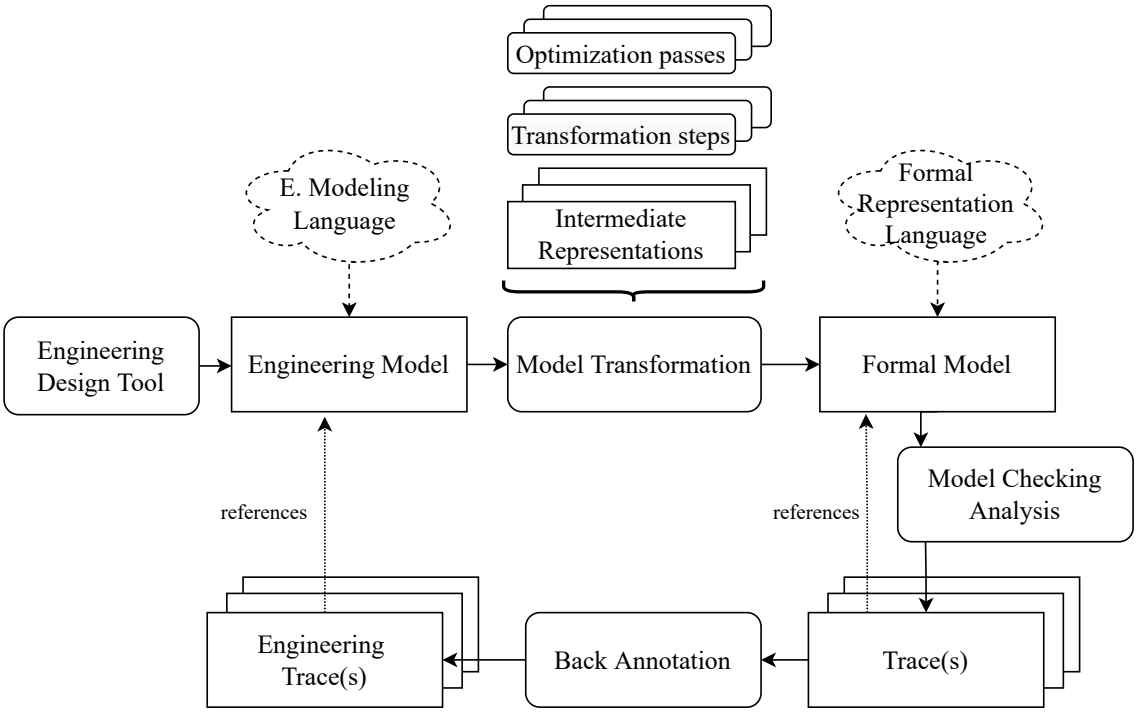



**Figure 3.1:** Typical process of a model checking toolchain.

**Backannotation and different Trace representations** If the tool finds any issues, it might return a counterexample or counterexamples as execution traces of the formal model. This has to be backannotated to the original engineering model to help the user mitigate the issue found.

## 3.2 Problem Statement

As it was shown in Section 3.1, using formal methods includes a complex verification process. Furthermore, error properties also have to be designed and added to the model checker, the right configuration has to be found and so on. All in all, it takes a considerable amount of effort to use these tools, but in a lot of cases it is worth it for the mathematically proven results.

However, the introduced process had to be implemented in the verification toolchain and it might contain different issues due to human error. Therefore the main question motivating my research was:

**How can we trust formal verification tools?**

If we do not make sure that all the implemented steps in the verification process are correct, the results of the tool are basically invalid: both false positives (i.e. false alarms) and false negatives (i.e. missed bugs) can possibly happen and thus the advantage of getting proofs with mathematical precision is lost.

**Model Checking Algorithms** The core part for verification is the analysis executed by the model checker. This analysis is not just for finding potential issues, but also to

prove soundness: if it finds no issues regarding the error property, ideally we expect that there really is none [27]. A lot of work goes into the correct formalization of the algorithms used in the analysis and also to proving that they are correct [25].

**Model Transformations** On the other hand model transformation steps, including optimizations, are usually much less rigorously checked, even though bugs in these steps can cause both false positive and false negative results.

For example, an issue in the model transformation step practically means that the analysis is reasoning upon a different model, over a different state space. Such an issue is really hard to uncover. If it causes a false alarm, it is possible to discover that the root of the issue is the model transformation and to debug it through the incorrect counterexample, but it requires a deep understanding on how the model transformation step works. However if the result is a false negative, it can easily remain completely hidden and the missed bug will remain in the model, even though the user will believe that the model is correct.

## 3.3 Challenges of Semantics in Model Transformation

**Semantic Gap** What makes model transformation steps more error prone is the *semantic gap* inbetween engineering modeling languages and formal representations. Although there are more and more initiatives for formalizing the semantics of engineering models [53, 54], *full formalization* of any engineering modeling language used in practice is *impractical*. On the other hand formal models are *fully unambiguous* mathematical models. Thus mapping an engineering modeling language to a formal representation requires the mapping of an ambiguous language to an unambiguous one.

**Advantages of Ambiguity** These modeling languages are made to enable the modelers to *design complex systems* with ease, thus they include complex language elements (e.g. non-determinism, concurrency, variables representing data). These *complex language elements* serve to enable many possible executions of the model while the model itself stays concise. These models are also often created iteratively with gradually increasing refinement, therefore these languages by design have to be able to express models that are still ambiguous and will only be refined in later iterations.

**The Need for Unambiguity** On the other hand, the precise reasoning of model checkers requires *unambiguous formal models*. This requires the developer implementing the transformation to make decisions regarding *missing and ambiguous parts* of the semantics of the engineering models. The correctness of these decisions depend on the developer's understanding of semantics. If semantics are misunderstood and bugs are introduced to the model transformation, the results of the model checker might become *invalid* and this might *not even be detected*.

### 3.3.1 Example of Ambiguous Semantics

One of the most well-known behavioral engineering models used in embedded systems are the different state machines ranging from simple finite state machines (FSMs) to UML or SysML [34] state machines. While the former is a low-level mathematical automaton, the latter offers more elements, such as variables to be able to express complex systems.

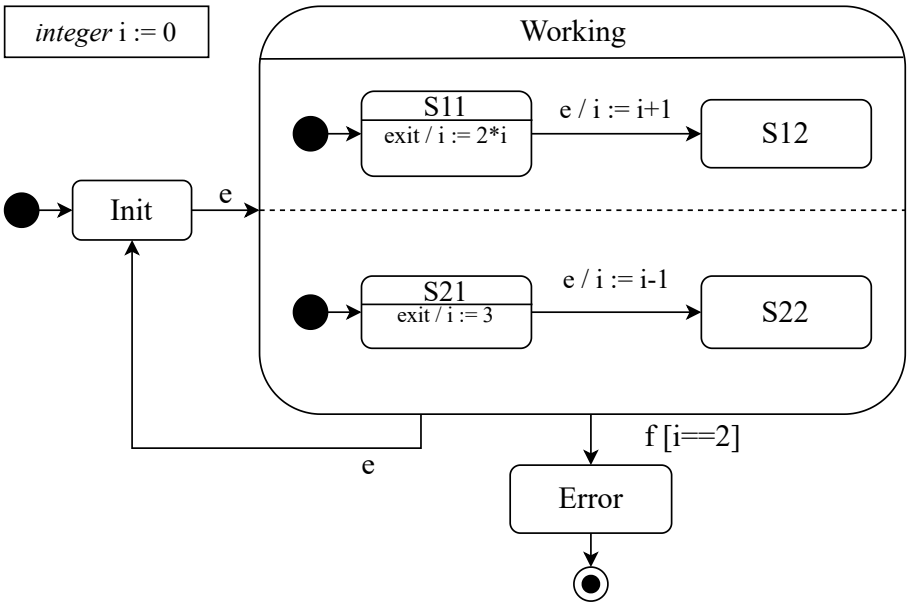



**Figure 3.2:** State machine containing language elements with ambiguous semantics.

The state machine shown in Figure 3.2 contains several typical language elements where interpretation of semantic rules highly affect the number of possible executions. Many state machine languages introduce concurrency in the form of orthogonal regions and non-determinism by adding conflicting triggers on several transitions. Another common addition are variables, used in actions and guards as well.

To be able to decide on the enabled executions, we have to precisely answer all of the following questions:

- Is full concurrency enabled, i.e. can the outgoing transitions of $S11$ and $S21$ fire in any order?
- Is the firing of a transition in a single region atomic, i.e. can anything else be embedded inbetween the execution of the exit action and the effect of the transition?
- Is there transition priority and if there is, what parts have priority, i.e. the outer or the inner transitions? Is it even possible to fire the transitions inside the composite state or will the model always go back to the *Init* state instead?

Different semantics and standards have different answers to these questions or some of them might even be configurable in some tools (e.g. transition priority). Furthermore, usually it is also possible to find questions that the semantics of a given language do not even answer unambiguously.

## 3.4 An Approach to E2E Validation of the Verification Process

The last two sections described why it would be necessary to validate the semantical mapping inbetween the engineering and formal model and why it is a difficult and complex task. Automatic validation is practically impossible due to the lack of fully formalized semantics of the engineering models.

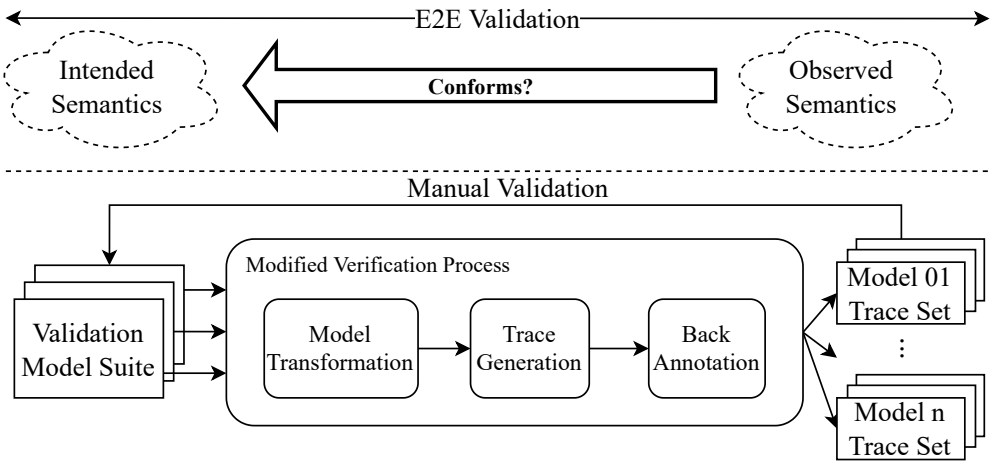



**Figure 3.3:** End-to-end (E2E) validation with trace generation.

I propose an approach providing the possibility of the end-to-end (E2E) validation of this process by utilizing the model transformation and the verification tool for trace generation. The goal of this validation process is to compare the intended semantics of the engineering model to the observed semantics after model transformation.

The trace generation algorithm shown in Figure 3.3 takes a model and uses the same process as it would use for verification (Figure 3.1), but where the model checker would execute the analysis, it generates a set of traces instead, which guarantee some kind of well-defined coverage or completeness. The trace generation algorithm is formalized and introduced in detail in Chapter 4.

The traces enable the user to manually compare the model and the execution traces, looking for executions that should not be permitted or a lack of traces that should. As input models for trace generation a validation model suite shall be designed, which covers a wide range of modeling elements and combinations of these elements. If this is accomplished, the generated traces might be able to uncover a wide range of possible issues in the different transformation and optimization steps or even in the back annotation process.

### 3.4.1 Another Use Case: Mitigating Modeling Mistakes

If the validation is deemed to be complete, there is another possible use case for the trace generation algorithm. The same process (Figure 3.3) can also be executed on a real-world engineering model instead of a test model.

The traces of a real-world model are useful if the modeler is unsure or might be mistaken about semantics. In this case the manual validation step shown on Figure 3.3 should be carried out by the modeler, this time comparing the modeler's understanding of semantics to the semantics implemented in the verification toolchain.

# Chapter 4

# Abstraction-based Trace Generation Algorithm

In this chapter I introduce a trace generation algorithm intended to be built around abstraction-based tools. First I describe the prerequisites of the algorithm (Section 4.1). After that the algorithm without abstraction and its extension with abstraction are formalized and explained (Section 4.2 and 4.3). The rest of the chapter adds an analysis on coverage guarantees and usability.

## 4.1 Prerequisites of the Trace Generation Algorithm

The algorithms introduced below were designed to be built around abstraction-based [26] tools which are capable of traversing abstract state spaces. Inevitably, some assumptions have to be made about how these tools work.

### 4.1.1 Abstraction Capabilities

Abstract state space traversal features building abstract reachability graphs on different abstraction levels and abstract domains. Thus the following requirements are established:

**Abstract Domains** The tool should include an *explicit-value abstract domain [12]* (or any other domain capable of representing concrete values for the variables of the model).

**Building ARGs** The tool should be able to build a *fully expanded ARG [13]* from a given formal model and precision and should be able to *concretize abstract traces*.

If these requirements are already fulfilled by the tool, it should not be difficult to implement the trace generation algorithms. If not, case-by-case modifications are also worth considering, e.g. it might be possible to modify the algorithm to work with some other abstract domains as well.

There is however a further tool specific design point which has to be considered together with the requirements for the usability of the trace generation.

**ARG semantics** ARGs [13] provide a fairly low-level graph structure for representing state spaces. It receives semantic meaning from the abstract states, operations and transfer function used, which will differ for each formal representation and tool.

For example, the granularity of abstract states can differ (e.g. are they only stable state configurations or are there abstract states representing unstable state configurations inbetween). What successors are calculated for the abstract states can also differ (e.g. if input events that do not change the current state configuration are taken into account or not, i.e. events which trigger no enabled transitions).

Thus the implemented logic and semantics for ARG building have to be compared to the desired goal with trace generation. If there is a mismatch between the two, slight modifications might be needed in the trace generation algorithm, such as filtering out some unnecessary states or traces during trace generation. An example for this will be shown in the case study used for evaluation in Section 5.1.2.3.

## 4.2 Generating Traces without Abstraction

Algorithm 4.1 utilizes the ARG building features of the abstraction-based tool. It uses an explicit-value domain and adds every variable to the precision and then builds a fully expanded ARG out of the input formal model. This will force the tool to build an ARG with the least possible abstraction, which should result in a reachability graph (RG) instead.

The main idea behind the algorithm is utilizing structural properties of reachability graphs. Reachability graphs are often finite: if the variables in the model have a finite range of possible values and the loops in the model have a finite number of possible states (i.e. at some point a state in the loop can be covered by one from earlier).

For all these finite reachability graphs Algorithm 4.1 will generate a finite set of traces. The generated reachability graph will also automatically assure that we do not unnecessarily repeat states in the traces – this will be illustrated by the example in Section 4.2.1.

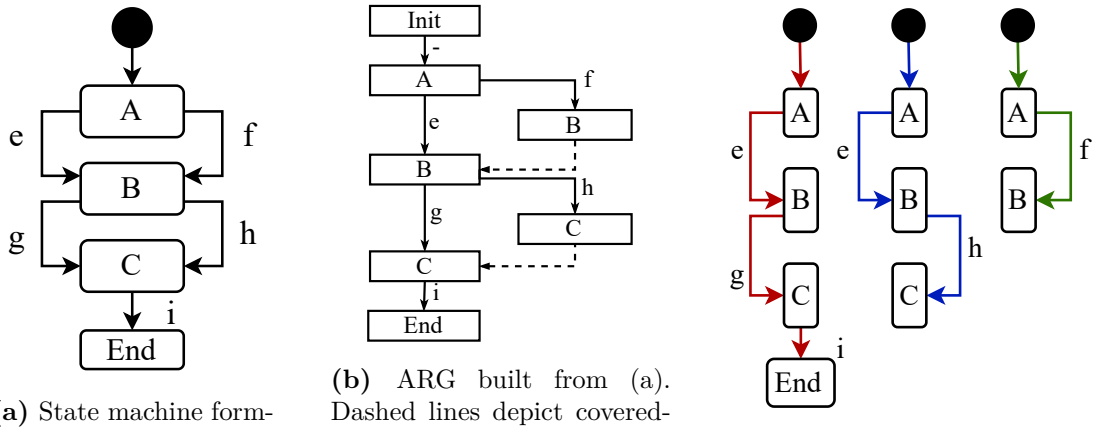
**Algorithm 4.1:** Trace generation algorithm without abstraction.

**input :** $F$: Formal model
**output:** $T$: Set of generated traces

1 $\pi$: Initial precision created including every variable
2 $ARG(N, E, C)$ := `buildARG(`$\pi$`,` $F$`)`
3 $n_0$ : initial node of ARG
4 $N_{leaf} := \forall n \in N$ where $n$ has no outgoing edge
5 **for** $\forall n$ *in* $N_{leaf}$ **do**
6 | $T \leftarrow$ trace from $n_0$ to $n$
7 **return** $T$

### 4.2.1 Trace Generation without Abstraction Example

In this section I will show how the algorithm works on a simple *state machine* from the model through the ARG to the traces. The *formal representation* created inbetween the model and the ARG is omitted, since the examples of this chapter are small and simple.

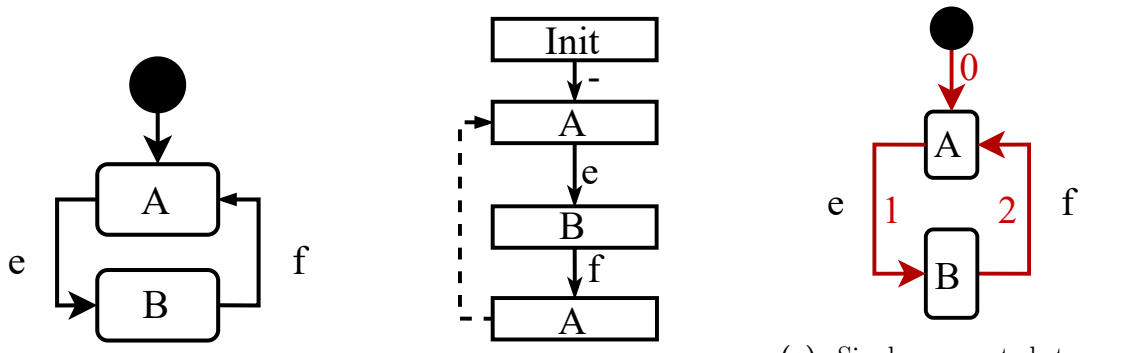



**(a)** State machine forming an "8"

**(b)** ARG built from (a). Dashed lines depict covered-by edges.

**(c)** Generated traces

**Figure 4.1:** Example showing the basic trace generation algorithm on a state machine.




**(a)** A state machine with a loop

**(b)** ARG for the state machine

**(c)** Single generated trace, transitions are numbered in the order of firing

**Figure 4.2:** Example showing how the algorithm avoids infinite loops

The input is visualized in Figure 4.1a. It is a simple state machine with 4 states and 5 possible incoming events.

In Figure 4.1b we depict a really simple *reachability graph*: there are no variables, so in this case the states of the graph represent the active state of the input model. The possible operations are assumptions on single incoming events, but the “assume” keyword was omitted for brevity.

In Figure 4.1c the result of the algorithm is shown. As the ARG is finite, the *set of traces* is also finite as well. The traces gradually shorten, because they stop at covered ARG nodes and will not re-explore the already discovered states – this ensures that the trace set remains relatively small and concise. This is also useful for the handling of larger models and loops – Figure 4.2 shows an example for the latter.

## 4.3 Utilizing Abstraction

Engineering modeling languages usually heavily utilize several variables of different types. Algorithm 4.1 keeps all of these variables in the precision, i.e. all variable values are explicitly tracked and are part of the abstract states. This might lead to a state space explosion in the ARG, which results in more and longer traces. These explosions are

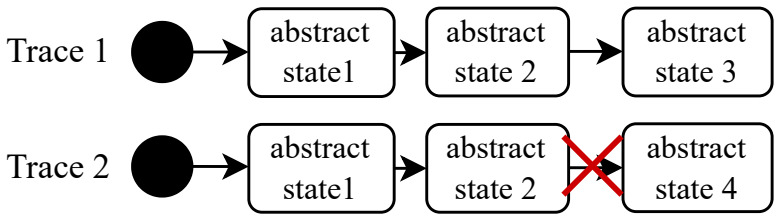



**Figure 4.3:** Trace 2 is only feasible if shortened, but then it is contained by Trace 1.

caused by the variables that are capable of holding many different values throughout the model's executions (e.g. indices and counters in loops).

Removing problematic variables from the precision mitigates such state space explosions. This is heavily utilized in verification and might be just as useful for trace generation as well – e.g. if we are mainly interested in possible control flows or possible values of other variables instead.

**Concretization and Feasibility Checks** Removing variables from the precision means that abstraction is introduced to the algorithm. Thus the algorithm has to be extended with feasibility checks and concretization (marked as `isFeasible(`$t$`)` and `concretize(`$t$`)`), as shown in Algorithm 4.2.

*Concretization* means creating a concrete trace out of an abstract one, if it is *feasible*. During concretization untracked variables are re-added to the trace and concretization finds a possible value for these (usually with the help of a SAT or SMT solver [27]).

**Infeasible Traces** Abstract traces that turn out to be infeasible will need special attention. The first important observation is that a shortened version of the trace might still be feasible. Finding the longest, still feasible part (marked as `shorten(`$t$`)` in the algorithm) is implementation specific. For example, it can be done with *interpolants* [8] or by just shortening the trace state by state and doing feasibility checks each time.

The possibility of generating shortened traces also necessitates a *filtering step* at the end. The reason for this is illustrated by Figure 4.3. *Trace 2* only becomes feasible if abstract state 4 is cut off. This shortened trace should be returned to show that abstract state 2 is reachable.

However if *Trace 1* was also generated then it would be confusing to return both, as *Trace 1* contains the shortened *Trace 2*. Thus in this case we should only return *Trace 1*. Note that the check for containment happens inbetween the abstract traces, as both could have several different concretizations.

### 4.3.1 Inappropriate Abstraction Level

There is another possible issue with infeasible traces which can be explained through Figure 4.3: it is possible that there will be no concretized trace leading to *abstract state 4*, even though in the concrete example it would be possible. For example, if *abstract state 4* is only reachable via an execution where a loop with an index $i$ has to be unrolled, but $i$ is not part of the precision, then the algorithm will find no trace leading to *abstract state 4*.

This issue can not be fully mitigated without adding $i$ to the precision, but the user might not want to do that, if adding $i$ slows down the execution too much. So instead of mitigation, a detection step is added to the algorithm.

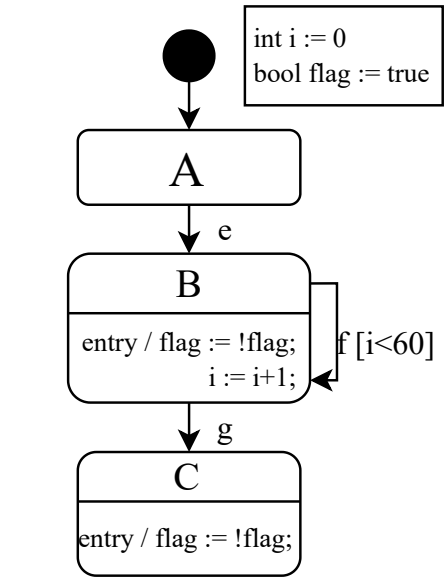
This step collects the abstract nodes that are pruned down (i.e. removed from the end) and also collects the abstract nodes that are concretized and included in the resulting traces. If, in the end, there is any node included in the former than is not in the latter then we could find no trace to reach that node with this abstraction, e.g. could not reach *abstract state 4*. This is reported in the output, so the user can decide whether a less abstract precision, e.g. adding $i$ to the precision, is worth trying.

**Algorithm 4.2:** Trace Generation Algorithm with Abstraction.

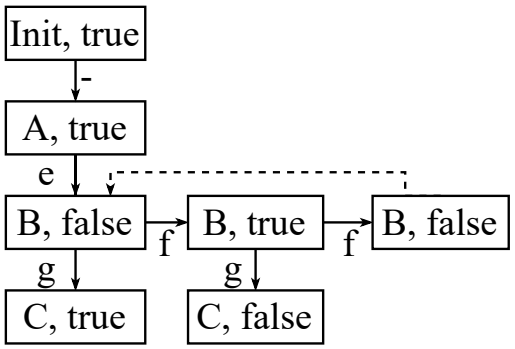
**input :** $F$: Formal model
1 $V$: Set of variables to be included in the precision
**output:** $T$: Set of generated traces, $fullCoverage$: True, if every abstract state is included in at least one trace
2 $\pi$: Initial precision created including every variable
3 $ARG(N, E, C)$ := `buildARG(`$\pi$`,` $F$`)`
4 $n_0$ : initial node of ARG
5 $N_{leaf} := \forall n \in N$ where $n$ has no outgoing edge
6 **for** $\forall n \in N_{leaf}$ **do**
7 | $T \leftarrow$ trace from $n_0$ to $n$
8 $T_{concrete} := \emptyset$
9 $N_{included} := \emptyset$
10 $N_{pruned} := \emptyset$
11 **for** $\forall t$ *in* $T$ **do**
12 | **if** $\neg$ `isFeasible(`$t$`)` **then**
13 | | $t' :=$ `shorten(`$t$`)`
14 | | $N_{pruned} \leftarrow \forall n \in t, \notin t'$
15 | **if** $|t'| > 0$ **then**
16 | | $N_{included} \leftarrow \forall n$ node of $t'$
17 | | $t_{concrete} :=$`concretize(`$t'$`)`
18 | | $T_{concrete} \leftarrow t_{concrete}$
19 **for** $\forall t_a, t_b \in T', |t_a| \leq |t_b|$ **do**
20 | **if** $t_a$ *starts with* $t_b$ **then**
21 | | $T_{concrete} := T_{concrete} \setminus t_a$
22 **if** $\exists n \in N_{pruned}, \notin N_{included}$ **then**
23 | $fullCoverage := False$
24 **else**
25 | $fullCoverage := True$
26 **return** $T_{concrete}, fullCoverage$

### 4.3.2 Trace Generation with Abstraction Example

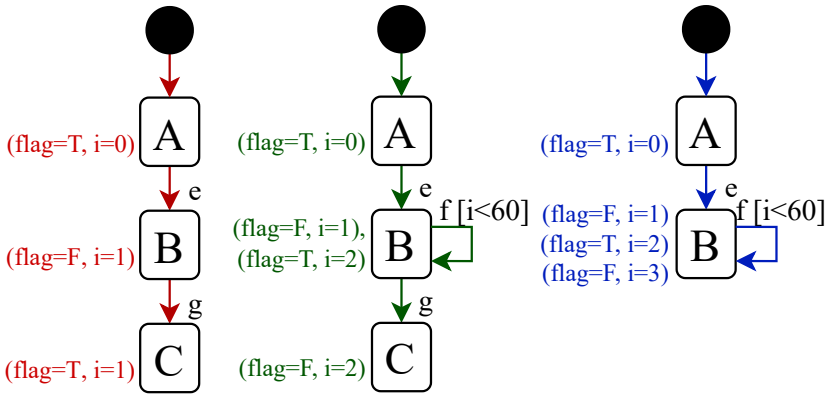
In Figure 4.4a a state machine with entry actions and two variables is shown. Using Algorithm 4.1 without abstraction would result in 60 traces, as all of the possible values of $i$ would be enumerated. If $i$ is removed from the precision, the ARG becomes much smaller as shown in Figure 4.4b and the algorithm results in the three traces shown in Figure 4.4c.



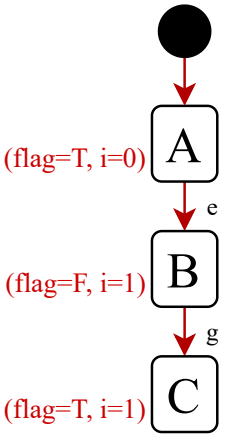

**(a)** State machine with two variables.




**(b)** ARG built by removing $i$ from the precision




**(c)** Resulting traces.




**(d)** Result if the guard is changed to $[i < 1]$

**Figure 4.4:** Example of using abstraction for trace generation.

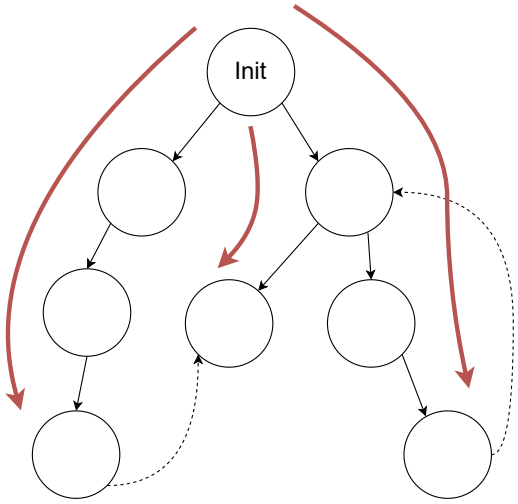



**Figure 4.5:** Possible ARG structure to illustrate ARG node coverage. The nodes are (abstract) states, the thick arrows show the abstract traces found in the ARG.

The difference between removing $i$ from the original model and removing it from the precision is highlighted on Figure 4.4d. If the guard of the input model is changed from $[i < 60]$ to $[i < 1]$, the self-loop of $B$ will not be able to fire at any time. As the change only concerns $i$, the ARG built by the algorithm stays the same. Yet the number of the traces goes down to one – this is due to the fact that the other two traces become infeasible and cannot be concretized.

**Inappropriate Abstraction Level** It is also worth to note that if we add, for example, the guard $[i == 50]$ on the transition leading to C in Figure 4.4a, we get an example where the algorithm will return false to the $fullCoverage$ value, as it does not unroll the loop without including $i$ in the precision and thus will not find a trace leading to $C, flag = true$ and $C, flag = false$. If we change the guard to $[i == 70]$, we will get the same warning, even though $C$ is not reachable in this case, but without the appropriate abstraction level, this can not be discovered, as explained earlier in Section 4.3.1.

## 4.4 Analysis of the Proposed Algorithm

In this section the coverage guarantees of the proposed algorithms are considered (Section 4.4.1). After that the strengths and weaknesses of the algorithm are also summarized (Section 4.4.2).

### 4.4.1 Coverage Guarantees

Coverage guarantees will first be considered on the level of the ARG and after that on a more general, engineering model level as well. The former is deduced from how the algorithm works, while the latter can usually be deduced from the former.

#### 4.4.1.1 Coverage on the ARG level

**State Space Coverage** Figure 4.5 illustrates why abstract state space coverage is accomplished: in the fully expanded ARG the whole abstract state space is represented by the ARG nodes (shown as circles). As there is a trace to every leaf (shown by the thick arrows) and a node is either a leaf or has outgoing edges, every node is included in at least one trace. Thus the algorithm easily covers the whole abstract state space. If there is no abstraction applied and all variables are included in the abstract states, the abstract state space coverage becomes concrete state space coverage as well.

In Section 4.3.1 the possibility of not reaching some of the abstract states was explained. If this happens then the abstract state space is not covered by the traces. Repairing the coverage requires finding the right abstraction, but the added complexity of this step can easily make the algorithm infeasible to use in practice. Instead the algorithm detects this incomplete coverage and lets the user decide on changing the abstraction level.

**ARG Edge Coverage** The edges leading inbetween nodes in an RG are also covered (excluding covered-by edges), as each ARG node is covered and each ARG node has exactly one incoming edge. For ARGs this coverage only holds if there are no infeasible traces amongst the abstract traces. Keep in mind that this does not guarantee any kind of coverage for the possible inputs, e.g. input events that trigger no response in the current state of the model might not be added as an edge while building the ARG.

**Comparison of Trace Generation with and without Abstraction** With the introduction of abstraction, an important trade-off appears, which is shown in Table 4.1. While without abstraction the whole concrete state space is covered, this can make the number of traces grow really high, making manual checks infeasible. On the other hand, abstraction is capable of mitigating state space explosion and can provide a concise set of traces, however untracked variables remain uncovered.

Both approaches have possible issues, which render them unusable for some models. Without abstraction the state space might explode so much, that the algorithm times out and does not even provide any traces. With abstraction one might not be able to find an appropriate abstraction level, which results in the loss of coverage guarantees.

| **Trace Generation** | **without Abstraction** | **with Abstraction** |
|---|---|---|
| **Abstract State Space Size** | can explode | can prevent explosion |
| **Number of Traces** | easily grows high | can stay concise |
| **Concrete State Space Coverage** | yes | no<br>*(does not cover untracked variables)* |
| **Possible Issues** | State Space Explosion<br>*(timeout)* | Inappropriate Abstraction Level<br>*(loses abstract state space coverage)* |

**Table 4.1:** Not using abstraction provides more guarantees, while abstraction helps keeping the number of traces concise, but still providing coverage for important variables.

These ARG-level coverages can be used to deduce what kind of coverages we might reach in the original input model, which is added in the next section.

#### 4.4.1.2 Typical Coverages for Engineering Models

Typical coverage criteria in engineering models, e.g. in conformance testing, do not build directly around state space. Thus it is important to examine coverage criteria on the input model for control and data flow as well, not just on the state space and ARG-level.

There are no general criteria for *behavioural engineering models*, rather separate, more specific definitions for state machines, activity diagrams and so on. But these are often similar in practice, as they build around *data and control flow*, which are present in all of these models. Thus typical state machine criteria are used here, but criteria for other model types is also possible to derive from these.

**Definition 4 (All-States Criterion [59, 61]).** This criterion is satisfied iff each state of the state machine is visited. ▪

**Definition 5 (All-Configurations Criterion [61]).** This criterion is satisfied iff each state configurations are visited (i.e. composite states and orthogonal regions activate several states at once). ▪

The satisfaction of All-Configurations implies satisfaction of All-States as well.

**Definition 6 (All-Transitions Criterion [24]).** This criterion is satisfied iff each transition in the statechart is traversed. ▪

**Definition 7 (Transition-Pair Criterion [52]).** This criterion is satisfied iff for each pair of adjacent transitions exists a trace that traverses the transitions in sequence. ▪

**Definition 8 (Decision Criterion [59, 61]).** This criterion is satisfied iff each guard condition is evaluated to true and false as well (if it is possible) in at least one trace. ▪

**Definition 9 (All-Defs Criterion [59, 61]).** Satisfied if for every defining action (variable value assignment) there is a trace which includes the defining action for a variable and at least one usage of that variable after the defining action, without the redefinition of the variable inbetween. ▪

**Definition 10 (All-Uses Criterion [59, 61]).** Satisfied if for every variable, every possible defining action and usage pair is covered in at least one trace, in definition-usage order, without the redefinition of the variable inbetween. ▪

The definitions do not take impossible to reach model elements into account, e.g. unreachable states are not included in the All-States criterion.

The relations between the coverages and the algorithms are shown on Table 4.2 and are explained in the paragraphs below.

**Loop Coverage** It is hard to find wide-spread coverage criteria specifically for loops, but it is still an important point to consider. While the ARG is built, loops are automatically unrolled until an already covered state is found. This ability guarantees that all possible *abstract states* in the loop will be covered, but infinite loops with repeating abstract states will *not prevent termination* either. Note however that variables excluded by abstraction will not be considered while unrolling. Moreover loops will be unrolled into *"lasso-shaped"* traces, i.e. the trace ends after the loop is unrolled.

| Criterion | Trace generation | | |
|---|---|---|---|
| | without Abstraction | with Abstraction | |
| | | No state space coverage violation detected | State space coverage violation detected |
| All-States | ✓ | ✓ | ✗ |
| All-Configurations | ✓ | ✓ | ✗ |
| All-Transitions | ✓ | ✓ | ✗ |
| Transition Pair | ✗ | ✗ | ✗ |
| Decisions | ✓ | ✓ | ✗ |
| All-Defs, All-Use | ✗ | ✗ | ✗ |

**Table 4.2:** Examining the algorithms regarding common state machine coverage criteria

**Data Flow Coverage** Traditional data flow coverage criteria (*All-Defs, All-Uses*) [57] do not hold, mainly because these coverage criteria do not take into account the values given to the variable at definitions. For example if there is a trace with a sequence of two definitions giving the same value to the variable and then a usage, the trace generation algorithm will not necessarily generate a trace that avoids the second definition. This is due to the fact that the abstract state regarding the variable will not change before and after the second definition.

However, this is actually a refinement of the criteria as this trace would be superfluous if we consider the possible values of the variables, not just the definition itself and this is exactly what the algorithm does.

**Trace Generation Algorithm without Abstraction** Due to the expanding of the whole state space, this algorithm covers most elements of the model: states and state configurations, transitions, guards (decisions). However, the combinations of these elements does not necessarily get covered, e.g. Section 4.2.1 and Figure 4.1a gave a good example of why it does not necessarily cover transitions pairs.

**Trace Generation Algorithm with Abstraction** As explained in Section 4.3.1 and shown in Section 4.3.2, it is possible in some cases with some specific loops that a model might not reach some abstract states which should be reachable.

This breaks the coverage of the abstract state space and thus will not guarantee the coverage criteria for the input model either. These coverage violations are detected, but can only be mitigated if the precision is changed.

However, for models where abstract state space coverage is intact, the input coverage criteria listed in Table 4.2 is guaranteed the same way as without abstraction as untracked variables do not directly influence these criteria, except *All-Defs* and *All-Use* which are not guaranteed in either case, as explained in the “Data Flow Coverage"" paragraph.

### 4.4.2 Usability and Feasibility for Validation

The main motivation behind the design of the algorithm was to enable the end-to-end validation of verification processes, mainly to validate the model transformation step (see Chapter 3).

Arguably the most important step regarding the feasibility of the validation is if the traces are appropriate and concise enough for the validating person to manually check.

**Appropriate Traces** The main question of the validation is what state configurations and values are possible during executions and in what ways are these possible to reach. Checking these should be feasible based on the coverage guarantees introduced above.

**Conciseness** Even when the whole of the concrete state space is considered, states are not visited repeatedly if it is not necessary (see Section 4.2.1). If the number of traces is still high, it is possible to filter out unimportant variables with abstraction and still have a good chance of keeping the coverage guarantees.

The points above are valid in many cases, but it also has to be mentioned that there will always be models, where validation is hardly feasible, e.g. if the variable causing a high amount of traces is important and cannot be omitted.

Timeouts are also possible if the (abstract) state space is too large (e.g. due to the sheer size of the model or state space explosion). Building ARGs by iteratively calculating successor nodes is typically done with SMT solvers [27], which can not be efficient in general, although they are well optimized in practice.

However contrary to these issues Chapter 5 will show that validation and other use cases are still feasible.

#### 4.4.2.1 Examples of Tools with the Necessary Prerequisites

In Chapter 5 a prototype implementation is introduced in detail, which was implemented in the toolchain of Gamma [50] and Theta [2, 58]. However, the algorithm would be possible to implement in other abstraction based tools as well. In this Section, a few other examples are introduced.

**CPAChecker** CPAChecker [11, 29] is a de-facto standard tool in software model checking. Most of its algorithms are abstraction-based and it utilizes ARGs and an explicit domain as well. The usability of the algorithm for software was not evaluated, but it would be worth considering.

**LoLA** LoLA [56, 62] is a low-level Petri net analyzer. These properties make it possible to implement the trace generation algorithm without abstraction in the tool. This can prove advantageous in validating either transformation processes from business and engineering models to petri nets or to validate the petri nets themselves.

**PLCverif – Theta** PLCVerif [46] is a frontend for verifying Programmable Logic Controller (PLC) programs, utilizing several backends, including Theta. As the algorithm is

already implemented in the generic core of Theta, only PLCVerif would need extensions to handle several traces instead of a single counterexample.

Usually the ideal candidates to implement the algorithm in a given tool are its developers, as they already have the necessary knowledge about the code base of the tool. But with sufficient documentation this is not by all means necessary.

# Chapter 5

# Evaluation

This Chapter starts with the introduction of the prototype implementation of the validation process from Chapter 3 and the trace generation algorithms from Chapter 4 (Section 5.1). Next, it details the goals and design process of this evaluation (Section 5.2). Section 5.3 elaborates on the design of the validation model suite, while Section 5.4 and Section 5.5 describe and discuss the results of both case studies of this evaluation.

## 5.1 Prototype Implementation

In this Section I detail the prototype implementation of the algorithms introduced in Chapter 4. This implementation is realized in the tools Gamma and Theta and shows how the prerequisites added in Section 4.1 apply to these tools.

### 5.1.1 Gamma and Theta

**Gamma** The Gamma Statechart Composition Framework [50] is an open-source modeling toolset with the goal of adding integrated verification and code generation features. It supports UPPAAL [44] and Theta [3, 58] as verification backends. It has a textual language for modeling, but it is also capable of visualizing models and traces with PlantUML [1]. It has been under active development since 2016, continually extending its features.

**Theta** Theta [3, 58] is a generic and highly configurable, abstraction-refinement based open-source model checker. It is capable of handling several formal representations (e.g Symbolic Transition Systems, Control Flow Automata, Timed Automata). It is mainly built around CEGAR [25], but is also capable of executing BMC and lazy abstraction. Furthermore, these basic algorithms can be further configured by changing the abstract domain, the refinement strategy, the SMT solver or some other parameters used in the chosen algorithm.

### 5.1.2 Process and Implementation

In this section the verification and the trace generation process of the Gamma-Theta toolchain is described to give an overview on how the trace generation algorithms become a

[1] `https://plantuml.com/`

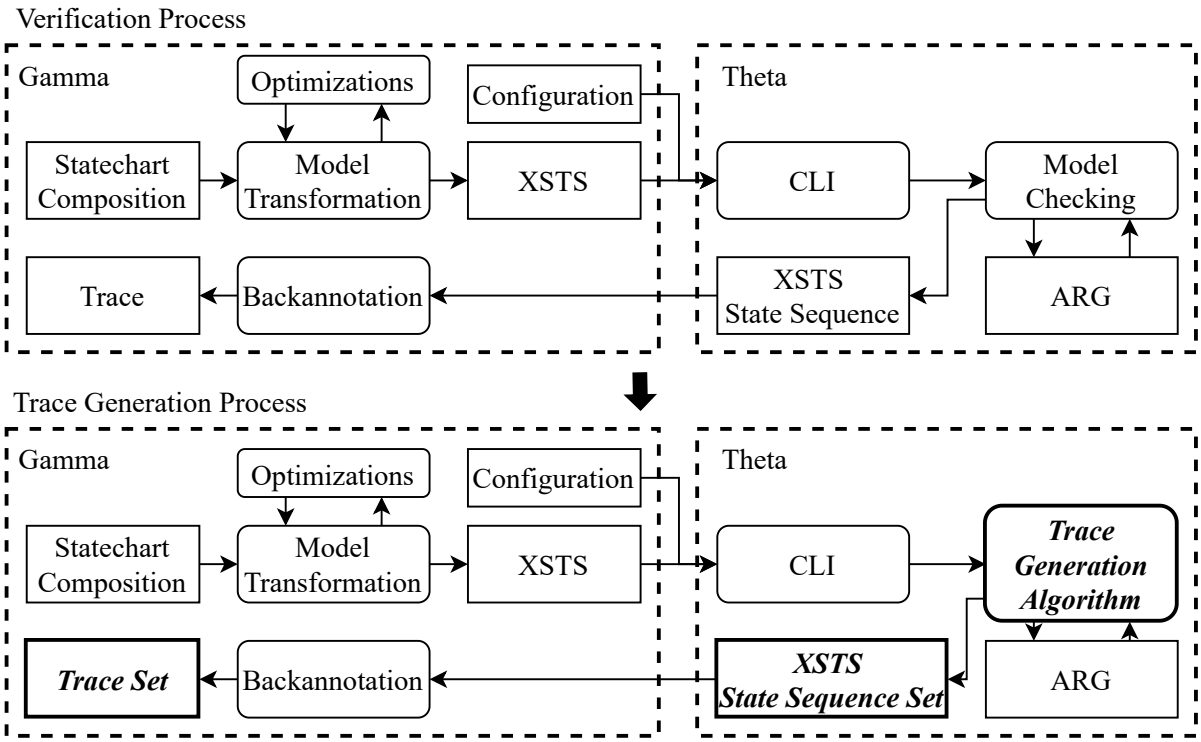



**Figure 5.1:** Verification and trace generation process

usable feature. The implementation specific details of the prototype of the trace generation algorithms are also detailed here.

#### 5.1.2.1 High Level View of the Process

The two tools together are capable of executing a complete formal verification process, shown at the top in Figure 5.1. The starting point of this process is a statechart or statechart composition, modeled in Gamma. This is transformed into a formal model, called eXtended Symbolic Transition System (XSTS) [51].

This is then given to Theta as the input model, together with the configuration of the analysis. We will consider the model checking analysis of Theta as a black-box for now – all we know is that it is building ARGs and returns a counterexample in the form of an XSTS state sequence when done.

Gamma is then capable of backannotating this state sequence, so it can be shown and visualized as a trace of the original statechart composition.

It is important to note how complex this process is, even though every step is essential for verification. There are 5 different models or formalisms that represent the model or some part of it:

- Gamma Statechart (composition),
- XSTS,
- ARG (nodes, edges, statements and actions),
- XSTS State Sequence,

- Gamma Trace Language.

Any of the transformations inbetween these representations can introduce issues and bugs.

The bottom process in Figure 5.1 is a modified version of the verification process. Instead of verification, it uses the trace generation algorithms from Chapter 4 and returns a set of traces instead of a single counterexample.

The prototype implementation is integrated into the configuration language of Gamma. This enables the user to easily add or remove variables if using abstraction and execute trace generation with only a few clicks in Gamma.

#### 5.1.2.2 Implementing Abstraction-based Trace Generation in Theta

The following points detail the tool-specific details, that were left as implementation specific in Chapter 4.

**Prerequisites** Theta more than suffices for the prerequisites detailed in Section 4.1. It is *abstraction-based* and the main structure used in the analyses is the *ARG*. It has configurable abstract domains, including the *explicit domain* and the initial precisions already have some possible configurations (i.e. empty precision or inclusion of all variables), which are easy to extend with a new one. Thus building fully expanded ARGs for arbitrary models and precisions can be easily done in the tool.

**Feasibility Checks and Concretization** These features are implemented with the help of SMT solvers in the tool, which are also capable of returning an interpolant, which can be used for shortening the trace.

**Configurability** The algorithm is implemented, so the algorithm can be used both with and without abstraction. If abstraction is chosen, a list of variables can be provided, which contains the variables that should be included in the precision.

#### 5.1.2.3 XSTS Specific Additions

Gamma transforms the state machines to a formal representation called eXtended Symbolic Transition System (XSTS) [51]. XSTS models represent everything with variables: not just the actual variables of the state machine, but control structures (i.e. states) and the input and output events as well.

This required small adjustments at several points during implementations, such as:

- If abstraction is used, the variables representing control structures and output events have to be automatically included in the precision.
- The transfer function allows branching into a direction where no transition is fired – these become small "dead-ends" (the unchanged state is immediately covered by its predecessor), which form traces that are unnecessary for validation, so these dead-ends are cut-off from the ARG.

These adjustments are simple additions, which were made with the main goal of validating model semantics in mind. Control structures and output events are both expected to

always be tracked. While in conformance testing it might be useful to check if a model is really ignoring input events that it should not react to, in this case this was already shown and presumed, thus traces only illustrating this behaviour would just make the number of generated trace unnecessarily high.

## 5.2 Evaluation Design

I designed this evaluation to assess the feasibility and usability of the trace generation algorithm (Chapter 4) and the end-to-end validation process (Chapter 3).

### 5.2.1 Research Questions

The evaluation was designed along the following research questions:

**RQ1** Is manual validation feasible based on the number and content of the generated traces?

**RQ2** What types of issues can the validation process uncover?

**RQ3** Can the trace generation be successfully executed on real-world models and give meaningful insights about behavior?

### 5.2.2 Process and Goal of the Evaluation

The research questions can be divided into two parts: RQ1 and RQ2 are concerned mainly with the end-to-end validation process, while RQ3 extends the scope to real-world models. Thus the evaluation can also be divided into two case studies:

**E2E Validation** This case study evaluates the model transformation validation process introduced in Chapter 3

**Real-World Models** Case study exploring trace generation on real-world models

#### 5.2.2.1 End-to-End Validation

The goal of this case study was to show that the validation process is capable of discovering inconsistencies and issues in the different steps (mainly the model transformation) of the verification process of Gamma and Theta. The generic process itself was already introduced in Chapter 3.

This case study also serves as the evaluation of the trace generation approach of this report, which is deemed satisfactory, if RQ1) the generated traces are appropriate for the manual checks, RQ2) the validation process uncovers different issues in the tools (or the lack thereof).

End-to-end validation consisted of the following steps:

1. Systematic design of a modeling suite for validation of semantics (Section 5.3)
2. Executing trace generation on the modeling suite, adding and changing abstraction as needed

3. Quantitative evaluation: size of models (states, transitions, variables), size of generated traces (number and lengths) (Section 5.4.1)

4. Qualitative evaluation: manual check of traces, discovering findings and issues, exploring explanations and solutions (Section 5.4.2)

#### 5.2.2.2 Real-World Models

This case study intends to give an example on the secondary use case of trace generation by executing it on real-world models from state machines of reactive systems.

There are different tutorial and industrial models in Gamma which are used for evaluation of new features on a regular basis. These are often complex systems of several state machines, but these state machines can also be used on their own.

Section 5.4.3 reports on the results of trace generation on models from three different projects: RQ3) investigates which ones were feasible to generate traces for and what insight this gave on Gamma and the models themselves. The goal was to see the limitations and capabilities of the algorithm on non-artifical models.

## 5.3 Designing a Validation Modeling Suite for Gamma

I systematically designed and modeled a validation modeling suite consisting of Gamma synchronous state machines for the E2E validation case study. The design followed iterative principles, i.e. it starts with a minimal set of model elements and progressively adds more and more elements to the models. This helps the manual validation process to find the roots of the discovered issues more easily.

The state machine language elements were extracted from the grammar of the Gamma statechart language [50] and then filtered based on the goal of the case study, i.e. it is out of scope to check every possible expression element (e.g. different operators) or state machine compositions.

The focus of this study are single, synchronous state machines, mainly concentrating on control flow and variables modifying control flow. The modeling suite covers a core set of the elements and semantic features of single, synchronous Gamma state charts. This core set can be arbitrarily extended in the future if needed.

The design went along the following groups of model elements in the following order:

A Basic elements (entry node, state, triggers)

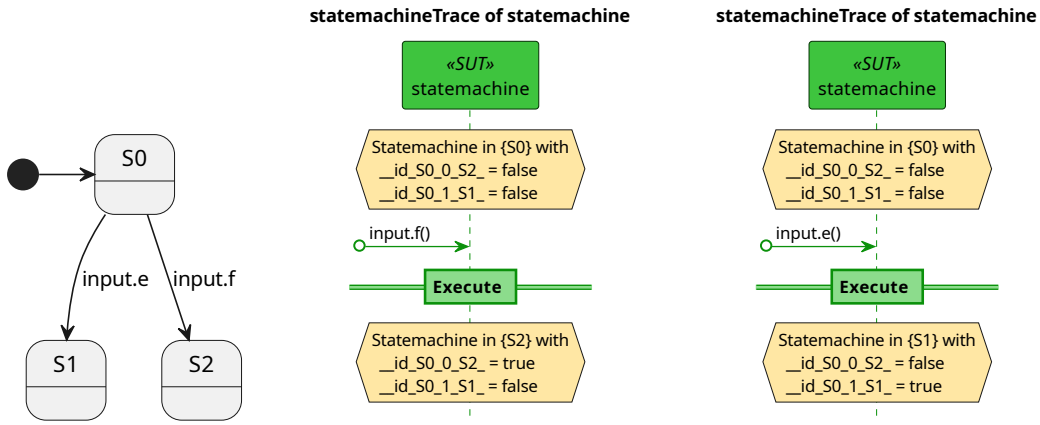
B Loops *(technically not a language element, but plays an important part in control flow)*

C Entry/exit actions, actions on transitions

D Composite States

E Orthogonal Regions

F Variables (assignments, modification of value, guards)




**(a)** Model 04 in package A. **(b)** First trace generated for the model. **(c)** Second trace generated for the model.

**Figure 5.2:** State machine and traces modeled, generated and visualized in Gamma.

The packages are built upon one another in the order shown above: each of these introduces a new group of model elements, while also utilizing the elements from previous packages in at least some of the models.

The order of the elements is based on their complexity and if they can be used without the packages before, e.g. everything depends on the basic elements, orthogonal regions utilize composite states and so on. Variables were added as last as they can be used to enable more possibilities and more complicated control flow to the models created earlier. In the end, 30 models were created.

### 5.3.1 Understanding Gamma Models and Traces

As the remaining part of the chapter will introduce a lot of Gamma synchronous statecharts and executions traces, this section will introduce how these traces can be interpreted.

Figure 5.2 shows model04, a simple state machine from the Gamma validation model suite I created. Executing trace generation results in two traces, also shown on the figure. The traces are visualized as sequence diagrams with a single life line, representing the model.

Input and output events are shown as lost and found messages. The environment and the model step alternately – the step of the model is represented by the *Execute* annotation on the lifeline.

After the model executed its step, the resulting state configuration and variable values are shown in the yellow hexagon. The state current configuration is in brackets. Keep in mind that in XSTS structure is also represented by variables and the boolean flags of transitions are shown in this hexagon as well, making it possible to see what transitions were fired during the step.

## 5.4 Results of the Case Studies

This section analyses quantitative results, such as the number and size of the generated traces (Section 5.4.1) and then the findings of the validation process are detailed (Sec-

tion 5.4.2). In Section 5.4.3 the results of the other case study on real-world models is reported.

### 5.4.1 RQ1: Quantitative Analysis of the Models and Traces

The basic trace algorithm was executed on all of the models, saving the resulting traces separately for each of them. For the traces with variables in package F, additional executions with abstraction were also added.

The qualitative summary of the traces is shown in Table 5.1. As the models are small, almost all executions ended up producing 1 to 4 traces with a maximum length of 4 (with exceptions in only three models: model18, model28 and model25).

Designing the whole model suite required a few additional iterations, where some "variants" of some already existing models were added, either to be able to check if some feature works as intended in both cases or to be able to understand findings more precisely.

Model01 uncovered a minor bug, where XSTS generation crashes for regions without transitions. The trace generation will be able to execute successfully, when this issue is fixed.

Time was not measured precisely, as all of the successful executions took mere seconds on a personal computer, which should be more than enough for general usability.

On the other hand, model 24 did not terminate without abstraction, even with a time limit of 30 minutes, so this execution was deemed to be a timeout. This was due to a self-loop incrementing an integer variable without a guard or other bound. The execution did not even finish building the ARG.

The manual part of the validation process did not cause much difficulty. The coverage guarantees introduced in Section 4.4.1 are also in order in the trace sets. Typically a model takes only a few minutes to check, mainly along the following guidelines:

1. if using abstraction, check if the tool reported a violation of abstract state space coverage,
2. check if the number of traces is approximately right,
3. check if the length of traces is approximately right,
4. check the state configurations and variable values in the traces,
5. check which transitions were fired, check guard values and executed actions as well.

**RQ1: Is manual validation feasible based on the number and content of the generated traces?**
All of the above mentioned quantitative properties were chosen to examine if the manual check of each trace set is a feasible task. The results show that this is true: there are only a few traces per model (if not, it can be mitigated using abstraction), which are also fairly short, but cover the reachable states and fireable transitions to show the behaviour of the model.

### 5.4.2 RQ2: Validation Findings

In this section the findings of the case study are reported. The rest of the traces and models will not be detailed in this report, but are available as an artifact [1].

| Model (Package A) | St. | Tr. | No. of traces | Max. length |
|---|---|---|---|---|
| model01 | 1 | 0 | - | - |
| model02 | 2 | 1 | 1 | 2 |
| model03 | 2 | 2 | 2 | 2 |
| model04 | 3 | 2 | 2 | 2 |
| model05 | 4 | 4 | 2 | 3 |
| model06 | 3 | 3 | 2 | 3 |
| model07 | 4 | 4 | 2 | 2 |
| model08 | 4 | 5 | 3 | 4 |

**(a)** Package A (with basic elements).

| Model (Package B) | St. | Tr. | No. of traces | Max. length |
|---|---|---|---|---|
| model09 | 1 | 1 | 2 | 3 |
| model10 | 2 | 2 | 2 | 3 |

**(b)** Package B (introducing loops).

| Model (Package C) | St. | Tr. | No. of traces | Max. length |
|---|---|---|---|---|
| model11 | 2 | 1 | 2 | 2 |
| model22 | 1 | 1 | 2 | 2 |

**(c)** Package C (introducing different actions).

| Model (Package D) | St. | Tr. | No. of traces | Max. length |
|---|---|---|---|---|
| model12 | 4 | 2 | 2 | 2 |
| model13 | 4 | 3 | 3 | 4 |
| model14 | 4 | 3 | 3 | 3 |
| model15 | 4 | 3 | 3 | 3 |
| model23 | 3 | 1 | 2 | 2 |

**(d)** Package D (introducing composite states).

| Model (Package E) | St. | Tr. | No. of traces | Max. length |
|---|---|---|---|---|
| model16 | 5 | 2 | 2 | 2 |
| model17 | 6 | 3 | 4 | 4 |
| model18 | 6 | 4 | 8 | 4 |
| model20 | 6 | 3 | 2 | 4 |
| model21 | 5 | 2 | 4 | 3 |

**(e)** Package E (introducing orthogonal regions).

| | | | | Basic | | Abstraction | |
|---|---|---|---|---|---|---|---|
| Model (Package F) | St. | Tr. | Var. | No. of traces | Max. length | No. of traces | Max. length |
| model24 | 1 | 1 | 1 | T | T | 1 | 2 |
| model25 | 1 | 1 | 1 | 1 | 11 | 1 | 2 |
| model26 | 1 | 1 | 1 | 1 | 2 | 1 | 2 |
| model27 | 2 | 1 | 1 | 1 | 2 | 1 | 2 |
| model28 | 3 | 3 | 2 | 60 | 62 | 3 | 5 |
| model30 | 1 | 1 | 1 | 1 | 3 | 1 | 3 |
| model31 | 2 | 1 | 1 | 1 | 3 | 1 | 3 |
| model32 | 5 | 2 | 3 | 1 | 3 | 1 | 3 |

**(f)** Package F (adds variables). Includes executions with abstraction (tracking no or boolean variables only).

**Table 5.1:** Summary of results on all models: number of states (St.), transitions (Tr.), traces (No. of traces) and the maximum number of state configurations in a trace (Max. length).

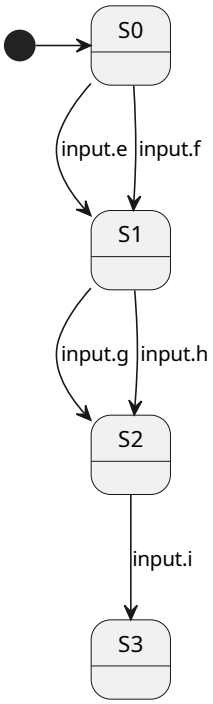



**Figure 5.3:** "Form 8" state machine modeled in Gamma (model 08, package A).

#### 5.4.2.1 Missing Default Values in XSTS

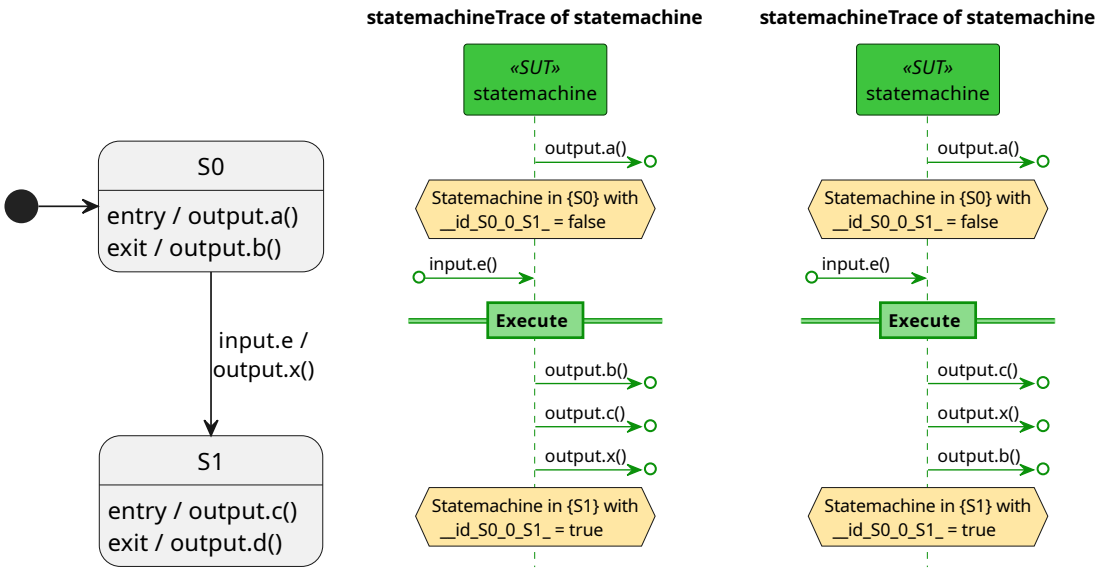
**Discovery** The first issue was found by discovering that a lot of traces are generated more than once when executing the algorithm. After checking the formal model and the ARG built for the model, a bug was found in the model transformation in Gamma.

**Explanation** As mentioned in Section 5.1.2.3, XSTS models [51] represent practically everything with *variables*, including boolean flags representing transitions. XSTS also represents the steps taken by the environment and the model separately and in an alternating manner. The transition variables are only relevant for the steps taken by the model and are set to false by default at the beginning of each step of the model, so that the relevant steps can be set to true later on in the step.

The hidden issue was that the value of the transition variables are also included in the environment steps, but were not set to false by default. Thus the abstract states produced by the environment steps superfluously reflected the transitions fired by the model earlier, creating abstract states that could not be covered by one another, even though in reality they should have been able to.

**Solution** The solution was to simply modify the model transformation step, so that it includes setting these variables to false in environment steps as well.

The size of the produced ARG is crucial in verification, as in larger models ARGs growing too big are often the cause for timeouts. This issue causes superfluous abstract states, letting the ARG grow larger than it should. To illustrate, the fully expanded ARG built for the small state machine shown in Figure 5.3 had 31 ARG nodes before fixing the issues and only 20 ARG nodes afterwards.




**(a)** model 11 introducing entry and exit actions

**(b)** Trace generated from model 11.

**(c)** Another "out of order" trace generated from model11.

**Figure 5.4:** model 11, package C. Expected order of actions is `b()`, `x()`, `c()`.

#### 5.4.2.2 Order of Operations inbetween Stable State Configurations

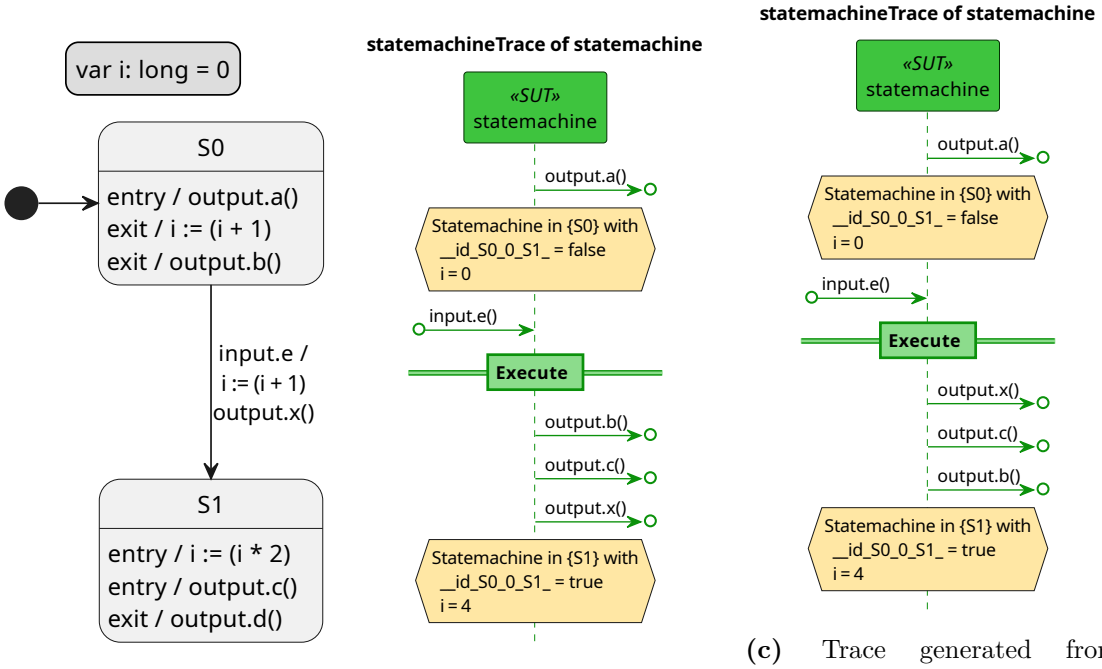
**Discovery** The naively expected semantics for the order of actions taken when a transition fires would be to execute the proper exit actions then the action on the transition and lastly the proper entry actions. Executing model 11 in package C it was discovered that the order of the output messages is often out of order and seemingly random, as shown on Figure 5.4. I could even generate different traces with some slight changes in configuration options that should not matter.

Furthermore, the execution of model 31 in package F (Figure 5.5) showed that the operations on variables seem to be consistent and correct in each case, even if the order of output messages are not right. If the ordering of the operations would not be right, $i$ could end up being either 3, 4 or 2 on the traces in Figure 5.5.

**Explanation** The intended semantics for synchronous state machines in Gamma is for the input and output messages to be handled like "signals", as they are part of a synchronous reactive system. The order these signals are changed in does not matter, only the value they have in the given step of the model.

But doing the same to variable operations would cause the model to be unintuitive and the result of the operations to be unambiguous, e.g. as in Figure 5.5a, so the variable operations were implemented to have a fixed and intuitive ordering.

**Solution** This finding is actually the result of a series of conscious decisions. The solution is simply to communicate these decisions with the user better in two ways: a) reflect the non-ordered nature of input and output events in the trace visualization by using parallel fragments, b) highlight the intended semantics in documentation more.




**(a)** model31 (package F, variant of model11)

**(b)** Trace generated from model31.

**(c)** Trace generated from model31 with slightly modified configuration.

**Figure 5.5:** model 31, package F

It is out of scope for this report to discuss if the intended semantics are suitable and consistent. Yet it might be worth to inspect from time to time if some decisions for semantics were added organically through implementation and if these were appropriate.

#### 5.4.2.3 Limitation of Parallel Executions

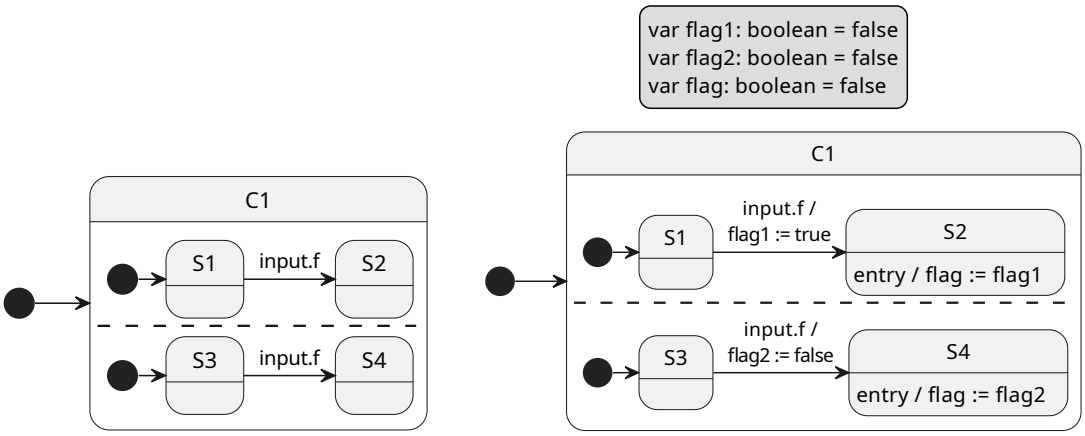
**Discovery** This finding stems from checking the traces generated by model16 and model32. The two models, shown in Figure 5.6, are variants of each other. Model16 belongs in package E, as it uses orthogonal regions, while model32 belongs in package F, as it extends model16 with variables.

The finding itself is that both models generated only a single trace (shown in Figure 5.7). This means that the state space checked by Theta does not include different orderings of entering states in these state machines.

This cannot really cause any issues in model16, as the resulting state configuration will be the same anyways and there are no variables. However, the trace shows that in model32 the possibility that we end up with $flag = true$ is not considered by the model checker.

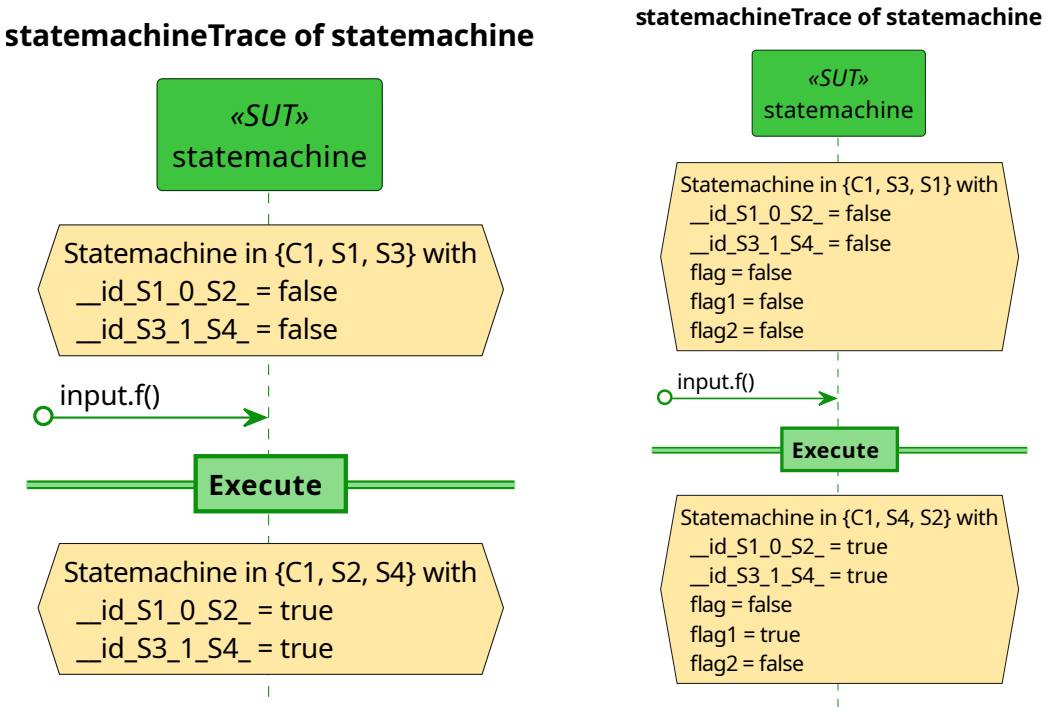
**Explanation** The semantical resolution for the issues in model32 in Gamma is that orthogonal regions are meant to be independent from each other. Gamma even issues a warning in the editor if instead of separate variables a single $flag$ variable is used: “Both this transition and the transition between S3 and S4 assign value to the same variable flag”.

However, actions in states are not checked for the same issue and the message above is simply a warning; it does not prohibit the modeler in creating the model this way.



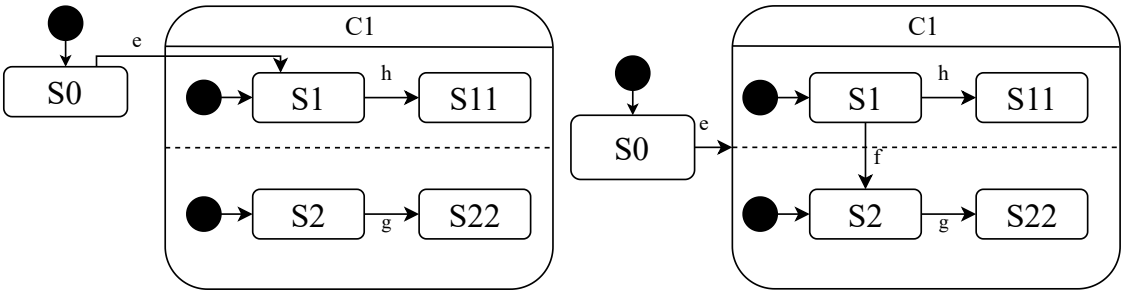

**(a)** model16 (package E) **(b)** model32 (variant of model16, package F)

**Figure 5.6:** model16 and model32, showing orthogonal regions




**(a)** Trace generated for model16. **(b)** Trace generated for model32.

**Figure 5.7:** Traces generated for the models on Figure 5.6




**(a)** model17 (package E) **(b)** model18 (package E)

**Figure 5.8:** Models with transitions going into orthogonal regions

The root of the issue is granularity: "temporary" state configurations inbetween "stable" ones are not considered. This "coarse" granularity is crucial when verifying larger models as otherwise verification might practically never complete due to ARGs becoming too large.

Thus when transitions are fired throughout several steps of the state machine, they behave as expected, i.e. if one of the triggers in model16 is changed to be different, Theta will include the state configurations S1, S4 and S3, S2 as well.

**Solution** As we have shown, for verification to be reliable, the orthogonal regions shall be modeled so that they are independent from each other. But this expectation should be communicated better with more warnings (or even prohibitions).

At this point it is worth to point out that the model suite of this case study can be used as a specification by example to help in communicating the presumptions and limitations discovered here or above in Section 5.4.2.2.

#### 5.4.2.4 Visualizing Transitions Crossing Composite states with Orthogonal Regions

**Discovery** There are several minor issues regarding transitions crossing borders of composite states and orthogonal regions.

**Explanation** Several models exhibited issues, which were traced back to different root causes.

**model17, model18** These models, drawn manually in Figure 5.8, cannot be visualized by PlantUML, as it prohibits transitions entering a state in an orthogonal region. However Gamma does not prohibit them and the trace generation can be executed successfully.

The intended semantics of model17 are the same as if the transition with the trigger $e$ would go to $C1$ instead. The generated traces have shown that this is what happens. However, model18 should be prohibited by Gamma.

Based on the traces, model18 is capable of achieving the $\{C1, S22\}$ state configuration, shown in Figure 5.9, while model17 behaves as if the transition going to $S1$ would be going to $C1$ instead.

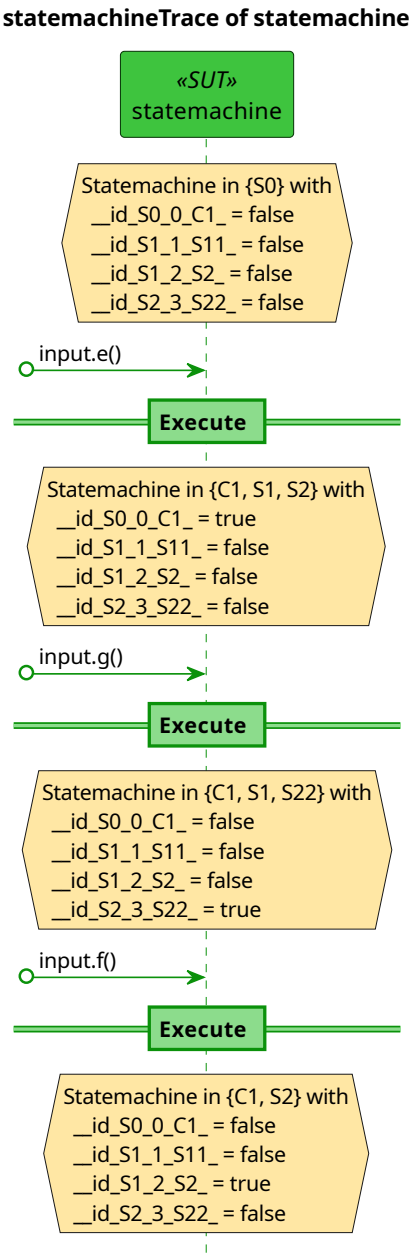
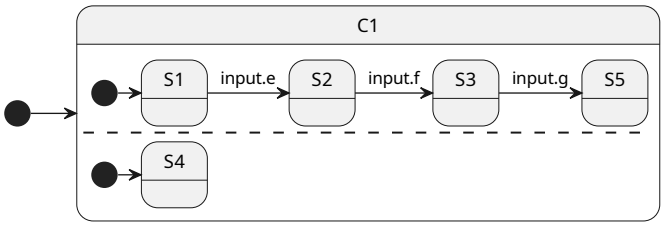
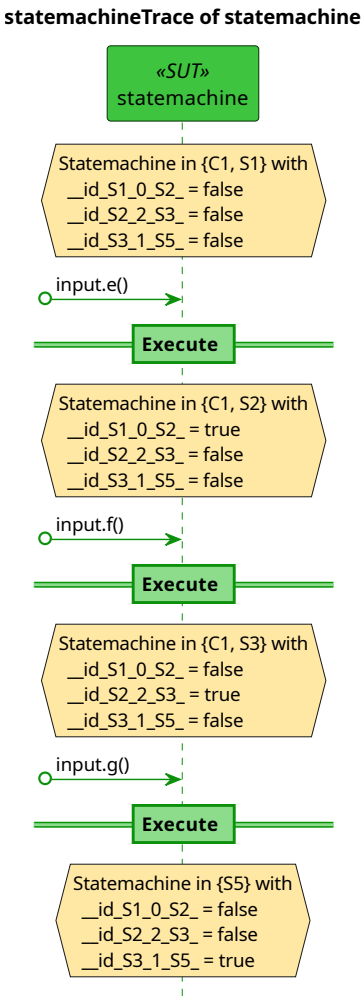



**Figure 5.9:** One of the traces for model18, enabling the model to reside in only one of the orthogonal regions




**(a)** model20 (package E), visualized incorrectly




**(b)** The single trace of model20

**Figure 5.10:** Incorrect visualization by PlantUML, uncovered by trace generation.

**model20** This model is incorrectly visualized by PlantUML, as based on the textual representation, $S5$ is not supposed to be a part of the composite state. But trace generation instantly reveals the issue by generating a correct trace and displaying the real possible state configurations as shown in Figure 5.10.

**Solution** For model20 this is a simple visualization issue, which should be debugged to display the models correctly. The importance of it comes from the ability to cause misinterpretation and confusion, especially in more complex cases.

Model17 also uncovers a visualization issue, but this time the feature set of PlantUML might not be able to cover like this and finding a solution for that will not be that simple.

For model18, there was a missing validation rule, as such crossing transitions should be prohibited by Gamma in the editor already. Although it was easy to fix, such validation rules play a really important part in mitigating modeling errors, e.g. typos. If a model like this is verified, the modeler ends up with a hidden invalid result.

| Model | Number of traces | Number of traces with no variables |
|---|---|---|
| **TrafficLightCtrl** | 21 | 10 |
| **GroundStation** | 10sec steps: 5, 5sec steps: 10 | 5 |
| **Spacecraft** | Timeout | 1 (incomplete coverage) |
| **Signaller** | Timeout | 12, only integers excluded: 33 |

**Table 5.2:** Result of trace generation on models from real-world examples.

**RQ2: What types of issues can the validation process uncover?**
The validation process was able to uncover several issues regarding model transformation, granularity and limitations of the formal representation (including missing executions) and visualization. It did not only uncover simple implementation bugs, but also *limitations of the generated models* that can easily invalidate verification results and require more than a simple patch of the tool.

### 5.4.3 RQ3: Traces of Real-World Models

So far the main use case introduced for the algorithm was the end-to-end validation of model transformations in the verification process. Another possible use case is uncovering mistakes in real-world models, as explained in Section 3.4.1.

Model developers utilizing the trace generation feature can gain insight on the possible executions of their model, uncovering misunderstandings in semantics, e.g. possible "corner-case" executions that the modeler did not think of or limitations of the verification the user did not know about, such as the one reported in Section 5.4.2.3.

Due to its inherent goal, the validation model suite contains only artificial models. To evaluate the usability of trace generation on real-world models, the prototype was executed on some models of the tutorials and industrial case studies [38] available for Gamma.

The result of the execution was checked on some synchronous state machines from:

- the Crossroads test/tutorial models[2],
- the signaller subsystem of the Railway Traffic Control System case study[3],
- and the Simple Space Mission case study[4].

Table 5.2 summarizes the results of the execution, while the results per model are detailed below.

**Ground Station** The Ground Station state machine is part of the Simple Space Mission case study and is shown on Figure 5.11. It contains two timers, which trigger some of its outer transitions.

[2] `https://github.com/ftsrg/gamma/tree/master/tests/hu.bme.mit.gamma.tests/model/Crossroads`

[3] `https://github.com/ftsrg/gamma/tree/master/examples/hu.bme.mit.gamma.railway.casestudy/model/COID`

[4] `https://github.com/ftsrg/gamma/tree/master/examples/hu.bme.mit.jpl.spacemission.casestudy`

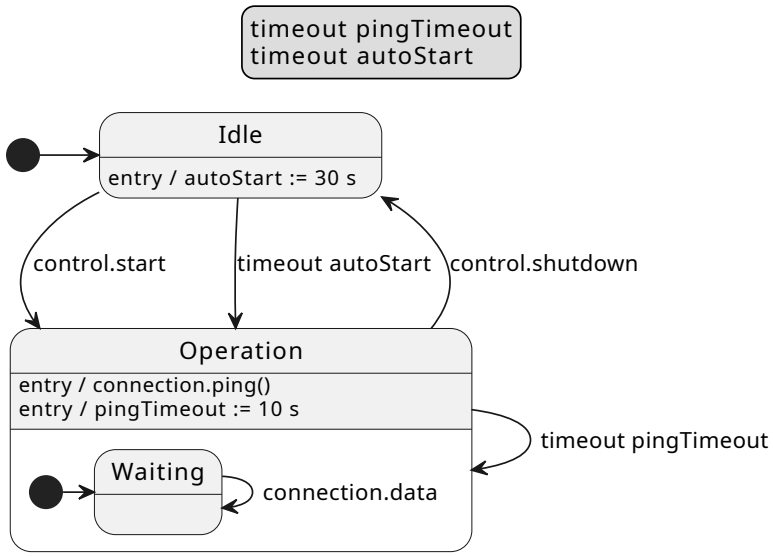



**Figure 5.11:** Ground Station model of the Simple Space Mission case study.

As shown in the second row of Table 5.2, different configurations result in a really different number of traces. The model transformation to XSTS requires a time step size to be set for timers. This will increment the relevant timers with this given step size each time before the model steps. As the outer transition of the *Operation* state has priority to the inner transition of $Waiting$, the inner transition is never fired if the time step is set to 10 seconds, but it is executed in some traces if the time step is smaller.

**Spacecraft** The other state machine from the Simple Space Mission case study is shown on Figure A.0.1. This model illustrates the limitations of this trace generation method.

Without excluding the *data* and $batteryVariable$ variables the execution never finished, while with abstraction the coverage of the state space becomes so low that only a single trace will be generated, as the loops decrementing these variables are not unrolled. This is detected and reported in the report file generated by Theta. This phenomenon is explained in detail in Section 4.3.1.

**Traffic Light Ctrl** This model on Figure A.0.2 depicts the state machine of a traffic light, capable of working in a normal or a blinking yellow mode. It is a common test model in Gamma and also includes some meaningless variables.

The number of traces here is higher than for the artificial models, but it is still feasible to check all of them, especially if the variables are not tracked.

**Signaller** Figure A.0.3 shows the Signaller state machine. This model features input and output events with boolean parameters, which significantly enlarge the state space of the model. It also features two integers as counters, which make abstraction essential, but contrary to the Spacecraft model, the abstract state space coverage is not violated here.

However, tracking the rest of the variables, which are either boolean or an enumeration (with 3 possible values) is feasible. While the number of traces here is fairly high, especially with some of the variables included, they are still feasible to look through, especially with a good understanding of the model.

**RQ3: Is trace generation capable of successful executions on real-world models?**
Trace generation was successful for most of the real-world models in the case study and these executions provided sets of traces appropriate for further manual analysis. The generated traces seem to be appropriate to illustrate how different aspects, like timers or priorities are handled and are capable of showing the relevant aspects right on the model in focus.
There are limitations as well: as in the case of the Spacecraft model, it is possible that a model has no "right" abstraction level, as with abstraction loss of state coverage is detected, without abstraction the trace generation will not terminate. Also, the number and length of traces might not scale well for some larger models and generate too many traces even with abstraction.

## 5.5 Discussion

**E2E Validation of Semantics** Based on the case study, the trace generation algorithm and the validation approach are deemed successful. The validation model suite did not completely cover the language elements of Gamma statecharts, but it includes a core set of these elements and can be easily extended. Determining what coverage should a validation suite should accomplish and designing a model suite sufficing to that would be a separate topic and thus this completeness was out of scope for this work.

Even then the validation approach was able to uncover several issues in different parts of the verification process: not just in the model transformation, but also visualization issues and limitations in concurrency and granularity. The validation suite and the traces can also serve in the tool's documentation as specification by example, informing the users about such presumptions in an intuitive way.

**Real-World Models** Although there are limitations in scalability and thus a time limit for execution is required, the trace generation can also be successful and useful on real-world models. It is capable of giving insight about semantics, such as priorities, right on the model itself.

**Threats to Validity** *Internal validity* is ensured by carefully following the steps of validation process. Trace generation was also re-executed on the models and produced the same input each time. Furthermore, the case studies' main goal was to show the feasibility and usefulness of the techniques (which was successful) not the exhaustive and complete validation of Gamma.

*External validity* is concerned with how well the results can be generalized. Different application domains of model checking have different aspects that make verification difficult and complex, while this case study is validating only a single modeling language. However some assumptions can be made on extending it to other domains and languages.

Checking *software code* will probably require some more work on scalability as the number and range of variables employed is usually much higher. However, for *other engineering models* (e.g. other state machine languages, activity diagrams, process diagrams) trace generation and the validation process will likely work in a similar manner as here due to their similarities.

# Chapter 6

# Related Work

## 6.1 The Landscape of Verification Tools

This work builds upon the abstraction capabilities of the tool under validation, thus it is important to examine how common abstraction is in such tools.

There are a wide array of formal verification tools available for many different application domains. Due to the need for comparative evaluation, many domains have benchmarking competitions at their disposal, showcasing the state of the art tools and techniques.

**SV-COMP [10][9]** The International Competition on Software Verification (SV-COMP) might be the largest of these competitions to date. It was specifically created for software verifiers and had 33 actively competing tools in 2022. Based on their report [10], 11 of these tools use Counterexample-guided Abstraction Refinement (CEGAR) [25], 8 use lazy abstraction, 9 of the tools use Explicit-Value Analysis and 5 use ARG-Based Analysis. There are several overlaps inbetween these properties.

**Model Checking Contest [43]** The Model Checking Contest (MCC) benchmarks verification tools on Petri net models. They had 7 actively competing tools in 2021. They do not have such a detailed report on the properties on the tools, but abstraction and explicit-value techniques seem to be present in the reported techniques of several tools [43]. One of the competitors is LoLA [56][62], which was already introduced in Section 4.4.2.1.

**Hardware Verification Competition** The hardware verification competition[1] executes benchmarks on hardware models mainly in the And-Inverter Graphs (AIG) format with 11 submitted tools in the last edition of the competition in 2020. Based on the report slides, some tools also apply abstraction here as well. For example nuXmv [21], one of the de facto standard tools, implements CEGAR and already has a feature called "computing reachable states"[2].

As shown in this section, abstraction and related techniques appear throughout all the different and domains in a significant amount of tools. While benchmarking competitions compare tools and give valuable feedback to tool developers, they are limited to only a few input model types and languages (e.g. C software, Petri nets).

[1] `http://fmv.jku.at/hwmcc20/#results`

[2] `https://usermanual.wiki/Document/nuxmvusermanual.465943104/html`

## 6.2 Test Generation with Model Checkers

Fraser et al. [33] describes several tools and papers about generating test cases with model checkers. Many of the works cited in this survey [31, 35] and even more recent works [45] differ greatly from my approach in that they use model checkers as black box tools, generating properties based on the test generation goals and feeding these properties to the tool as a verification problem, using the resulting counterexample as a test case.

In a subsequent paper, Fraser et al. [32] describes several drawbacks of this approach, e.g. a model checker might prioritise counterexamples that are easy to understand, but make no good test cases. This work states that model checkers could generate better quality test suites with some added techniques focusing on test generation (e.g. abstraction for testing, constraints and prioritization of counterexamples), i.e. not using the model checker as a completely black box tool.

Although my work generates traces with a different goal in mind, it relates to the realizations of these issues. When the model checker is seen as a black box, typically the whole verification process is utilized for the generation of a single test case, making several state space traversals necessary for the test suite. Instead, this work utilizes lower level features of the tool, such as ARG building, making the tool capable to generate all the traces in a single execution.

## 6.3 V&V of Model Transformations

There is a lot of available work on different approaches to the verification of different model transformations, such as UML state machines to colored petri nets [49], UML statecharts to Petri nets [60] or BPMN models to Petri nets [48], verifying properties, such as termination and structural properties.

Varró and Pataricza [60] fully verify several properties, such as syntactic correctness and completeness. For semantic correctness they give separate dynamic consistency properties, as semantic equivalence between the models cannot always be proved.

These approaches concentrate on automatically checking properties, while this work concentrates on with the validation of informal semantics, which cannot be fully automated due to the lack of formality. Chapter 3 explained why this lack of formal semantics is typical for many models and thus this report can be viewed as a complementary extension of the works mentioned above.

## 6.4 Conformance Testing of Different Tools and Compilers

Conformance testing is frequently used in practice. For example, the “Precise Semantics of UML State Machines (PSSM)” specification [53] defines execution semantics for UML state machines. It contains a conformance test suite containing state machines with execution traces, both modeled by hand. Any given execution tool that wants to conform to this specification must pass the conformance tests. Issues due to manual creation of traces include typos, inconsistencies on completeness and unambiguity as well [30].

Test generation and conformance test suites are commonly used in the testing of compilers [22]. Test generation most commonly builds on the grammar of the programming language. However, ambiguous or non-deterministic executions are rarely tested.

# Chapter 7

# Conclusion

## 7.1 Summary of Results

**Validating Semantics of Verifiers** Chapter 3 described the typical *formal verification process* and formulated the main motivation of this work (*"How can we trust formal verification tools?"*). It concentrated on why *model transformation* steps are prone to errors, especially when verifying engineering models. To counter such errors, I proposed an end-to-end validation process with execution trace generation at its core to detect issues in model transformation in formal verification tools.

**Trace Generation Algorithms** In Chapter 4, I proposed a novel algorithm for generating execution traces with model checkers. This algorithm is capable of utilizing abstraction to keep the trace set concise and thus make it feasible to check the traces manually during the validation process. I also gave detailed analysis on coverage guarantees, usability and comparison of trace generation with and without abstraction.

**Evaluation** Lastly, in Chapter 5 I implemented a prototype of the algorithms and the validation process and designed two case studies. The goal of the first one was to evaluate not just trace generation, but the whole end-to-end validation process. For this, I designed a validation model suite covering a core set of the state machine language of Gamma. After executing trace generation, I listed my findings in this chapter as well, including several different issues from minor bugs to hidden limitations. The models and the generated traces are available as an artifact for this report [1].

The other case study was created to investigate another use case of trace generation: generating traces for real-world models to discover modeling mistakes. For this I took some real-world models from earlier Gamma case studies and tutorials and checked what insights the generated traces can give on these models.

## 7.2 Future Work

As detailed in Chapter 4, Section 4.4.2.1, there are other tools in which the trace generation algorithm and the validation process would be possible to implement. Furthermore, Theta has several other frontends and formalisms, e.g. for C code and hardware models. These would also be interesting to employ trace generation in.

Implementing the algorithm in these tools would open the possibility of further experiments and case studies of validating model transformation and utilizing trace generation.

Issues with ambiguity are also typical for software (e.g. undefined behaviour in C), so execution trace generation might also be useful in software model checkers – however this might require further research into the abstraction aspect of the algorithm as variables play an even more prevalent role in software code.

Another interesting part to extend the trace generation algorithm itself would be to find ways of employing other abstract domains (e.g. predicate abstraction) for trace generation. This might prove useful if there are variables that cause state space explosion, but they are important and should not be completely ignored. Predicate abstraction might offer a solution, as it can represent predicates, e.g. statements about the possible values of the variable in a more compact way.

**Acknowledgment** Supported by the **ÚNKP-22-2-I-BME-205** New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

# Bibliography


[1] Zsófia Ádám and Zoltán Micskei. Abstraction-based trace generation to validate semantics of formal verifiers: Validation model suite, 2022. URL https://zenodo.org/record/7263707.

[2] Zsófia Ádám, Gyula Sallai, and Ákos Hajdu. Gazer-Theta: LLVM-based Verifier Portfolio with BMC/CEGAR (Competition Contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 433–437, Cham, 2021. Springer International Publishing.

[3] Zsófia Ádám, Levente Bajczi, Mihály Dobos-Kovács, Ákos Hajdu, and Vince Molnár. Theta: portfolio of cegar-based analyses with dynamic algorithm selection (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13244 of *LNCS*, pages 474–478. Springer, Cham, 2022. DOI: 10.1007/978-3-030-99527-0_34.

[4] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain Types: Abstract-Domain Selection Based on Variable Usage. In *Hardware and Software: Verification and Testing*, pages 262–278. Springer International Publishing, 2013. DOI: 10.1007/978-3-319-03077-7_18.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008. ISBN 978-0-262-02649-9.

[6] Levente Bajczi, Zsófia Ádám, and Vince Molnár. C for yourself: Comparison of front-end techniques for formal verification. In *2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. IEEE, 2022. DOI: 10.1145/3524482.3527646.

[7] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45319-2.

[8] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. DOI: 10.1007/978-3-319-10575-8_11.

[9] Dirk Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-72013-1_24.

[10] Dirk Beyer. Progress on software verification: SV-COMP 2022. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402, Cham, 2022. Springer International Publishing.

[11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[12] Dirk Beyer and Stefan Löwe. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Fundamental Approaches to Software Engineering*, pages 146–162. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37057-1_11.

[13] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007. DOI: 10.1007/s10009-007-0044-z.

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

[15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 326–337, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. DOI: 10.1145/2950290.2950351. URL https://doi.org/10.1145/2950290.2950351.

[16] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 326–337, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. DOI: 10.1145/2950290.2950351. URL https://doi.org/10.1145/2950290.2950351.

[17] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. Tests from witnesses. In Catherine Dubois and Burkhart Wolff, editors, *Tests and Proofs*, pages 3–23, Cham, 2018. Springer International Publishing. ISBN 978-3-319-92994-1.

[18] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer Berlin Heidelberg, 1999. DOI: 10.1007/3-540-49059-0_14.

[19] Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems*. Springer Berlin Heidelberg, 2005. DOI: 10.1007/b137241. URL https://doi.org/10.1007/b137241.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, June 1992. DOI: 10.1016/0890-5401(92)90017-a.

[21] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08867-9.

[22] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), feb 2020. ISSN 0360-0300. DOI: 10.1145/3363562. URL https://doi.org/10.1145/3363562.

[23] Shengbo Chen, Hao Fu, and Huaikou Miao. Formal verification of security protocols using Spin. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, 2016. DOI: 10.1109/ICIS.2016.7550830.

[24] P. Chevalley and P. Thevenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 205–214, 2001. DOI: 10.1109/CMPSAC.2001.960618.

[25] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003. DOI: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[26] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. DOI: 10.1145/186025.186051.

[27] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018. ISBN 3319105744.

[28] Tom Coffey, Reiner Dojen, and Tomas Flanagan. Formal verification: an imperative step in the design of security protocols. *Computer Networks*, 43(5):601–618, 2003. ISSN 1389-1286. DOI: https://doi.org/10.1016/S1389-1286(03)00292-5. URL https://www.sciencedirect.com/science/article/pii/S1389128603002925.

[29] Matthias Dangl, Stefan Löwe, and Philipp Wendler. CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 423–425, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

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

[31] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 384–398, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-68519-7.

[32] Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403–1418, 2009. ISSN 0164-1212. DOI: https://doi.org/10.1016/j.jss.2009.05.016. URL https://www.sciencedirect.com/science/article/pii/S0164121209001137. SI: QSIC 2007.

[33] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw Test Verif Rel*, 19(3):215–261, 2009. DOI: https://doi.org/10.1002/stvr.402.

[34] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[35] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 146–162, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48166-9.

[36] Vahid Garousi, Michael Felderer, Çağrı Murat Karapıçak, and Uğur Yılmaz. Testing embedded software: A survey of the literature. *Information and Software Technology*, 104:14–45, 2018. DOI: 10.1016/j.infsof.2018.06.016.

[37] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83. Springer Berlin Heidelberg, 1997. DOI: 10.1007/3-540-63166-6_10.

[38] Bence Graics, Vince Molnár, and István Majzik. Integration test generation for state-based components in the gamma framework. Under review.

[39] Orna Grumberg, Doron A Peled, and EM Clarke. *Model checking*. MIT press Cambridge, 1999. ISBN 978-0-262-03883-6.

[40] Havva Gülay Gürbüz and Bedir Tekinerdogan. Model-based testing for software safety: a systematic mapping study. *Softw. Qual. J.*, 26(4):1327–1372, 2018. DOI: 10.1007/s11219-017-9386-2.

[41] ISO/IEC. *Conformance testing methodology and framework*, 1994. ISO/IEC 9646.

[42] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 547–550, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. DOI: 10.1145/581339.581406. URL https://doi.org/10.1145/581339.581406.

[43] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, , S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, , J. Srba, . Thierry-Mieg, A. Walner, and K. Wolf. Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php, June 2021.

[44] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. DOI: 10.1007/s100090050010.

[45] Daniset González Lima, Rauĺ E. González Torres, and Pedro Mejía Alvarez. Automatic test cases generation for C written programs using model checking. In *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1944–1950, 2021. DOI: 10.1109/CSCI54926.2021.00361.

[46] Ignacio D. Lopez-Miguel, Jean-Charles Tournier, and Borja Fernandez Adiego. Plcverif: Status of a formal verification tool for programmable logic controller. 2022. DOI: 10.48550/ARXIV.2203.17253. URL https://arxiv.org/abs/2203.17253.

[47] Azad M. Madni and Michael Sievers. Model-based systems engineering: Motivation, current status, and research opportunities. *Systems Engineering*, 21(3):172–190, 2018. DOI: 10.1002/sys.21438.

[48] Said Meghzili, Allaoua Chaoui, Martin Strecker, and Elhillali Kerkouche. Transformation and validation of BPMN models to Petri nets models using GROOVE. In *2016 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, pages 22–29, 2016. DOI: 10.1109/ICAASE.2016.7843859.

[49] Said Meghzili, Allaoua Chaoui, Martin Strecker, and Elhillali Kerkouche. Verification of model transformations using Isabelle/HOL and Scala. *Information Systems Frontiers*, 21(1):45–65, May 2018. DOI: 10.1007/s10796-018-9860-9.

[50] 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 *ICSE: Companion Proc.*, pages 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

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

[52] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *«UML»'99 — The Unified Modeling Language*, pages 416–429, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[53] OMG. *Precise Semantics of UML State Machines (PSSM)*, 2019.

[54] OMG. *Semantics of a Foundational Subset for Executable UML Models*, 2021.

[55] Mathias Preiner, Armin Biere, and Nils Froleyks. Hardware model checking competition 2020. 2020. website: http://fmv.jku.at/hwmcc20/.

[56] Karsten Schmidt. Lola a low level analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, pages 465–474, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44988-1.

[57] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A survey on data-flow testing. *ACM Comput. Surv.*, 50(1), mar 2017. ISSN 0360-0300. DOI: 10.1145/3020266. URL https://doi.org/10.1145/3020266.

[58] Tamas Toth, Akos Hajdu, Andras Voros, Zoltan Micskei, and Istvan Majzik. Theta: A framework for abstraction refinement-based model checking. In *FMCAD*. IEEE, 2017. DOI: 10.23919/fmcad.2017.8102257.

[59] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[60] Dániel Varró and András Pataricza. Automated formal verification of model transformations. In *Critical Systems Development with UML - Proceedings of the UML'03 workshop*, page 63, 2003.

[61] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt University of Berlin, 2010.

[62] Karsten Wolf. Petri net model checking with LoLA 2. In Victor Khomenko and Olivier H. Roux, editors, *Application and Theory of Petri Nets and Concurrency*, pages 351–362, Cham, 2018. Springer International Publishing.

# Appendix

The following pages contain the real-world models introduced in Chapter 5, Section 5.4.3, except the Ground Station model which was added directly to the chapter (Figure 5.11).

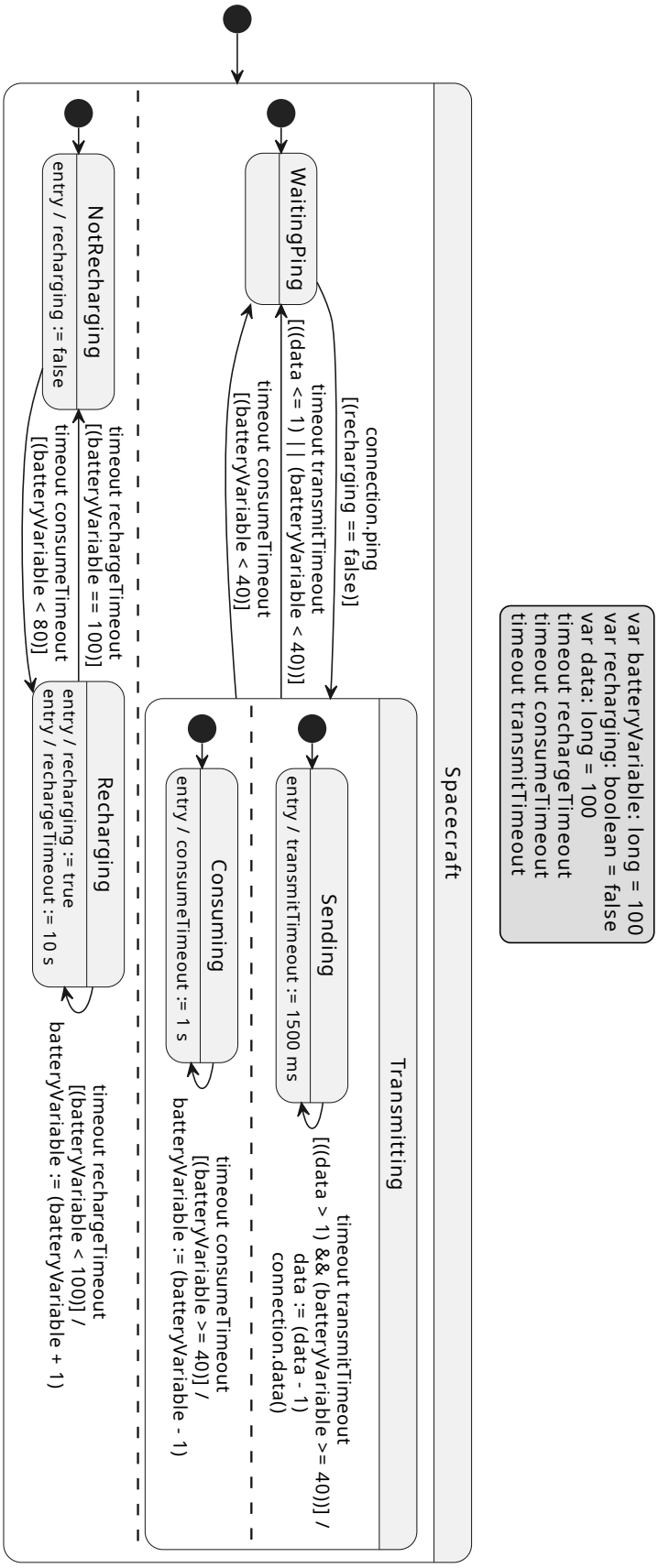



**Figure A.0.1:** Spacecraft model of the Simple Space Mission case study.

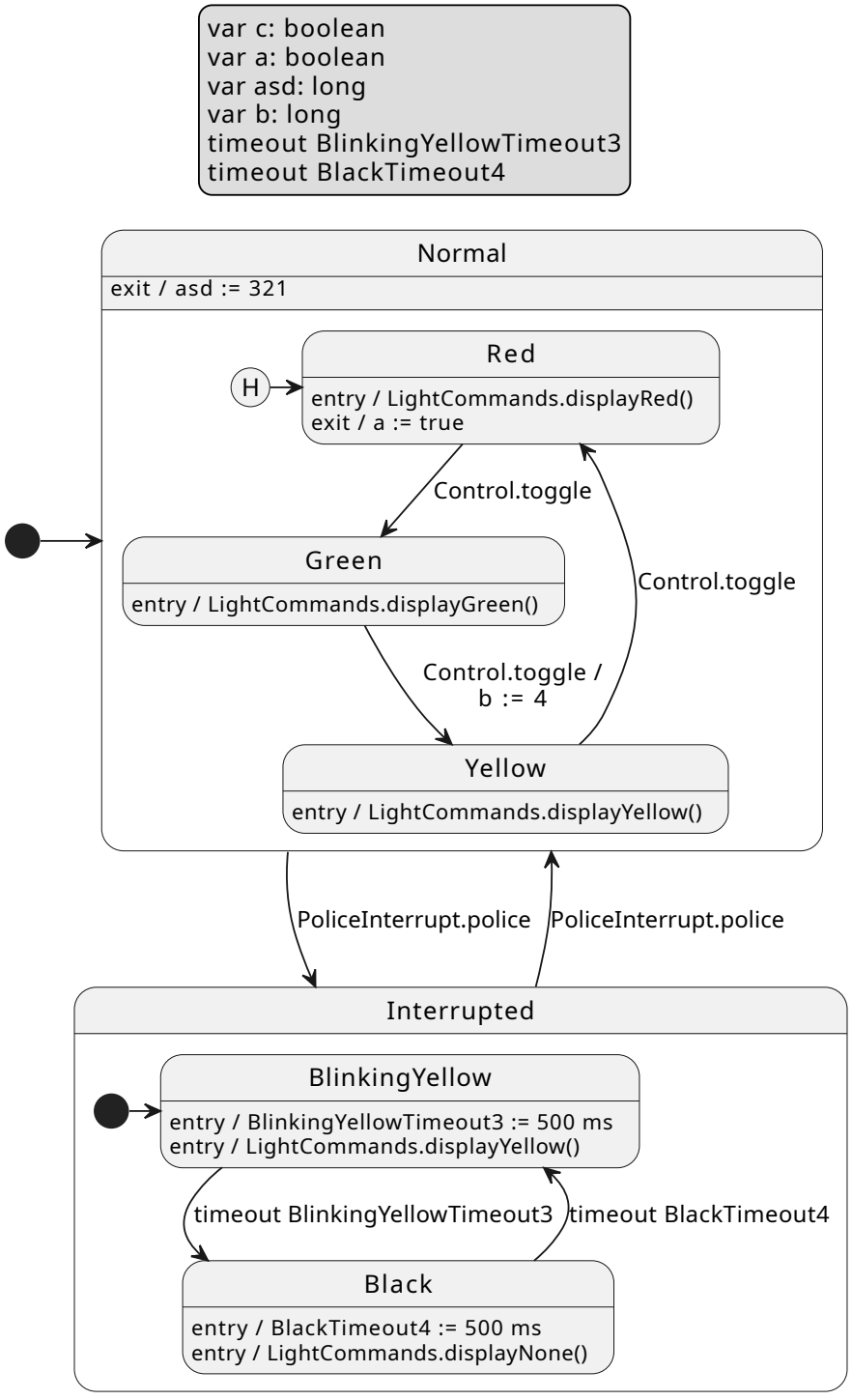



**Figure A.0.2:** Traffic Light model from the Crossroads tutorial models.

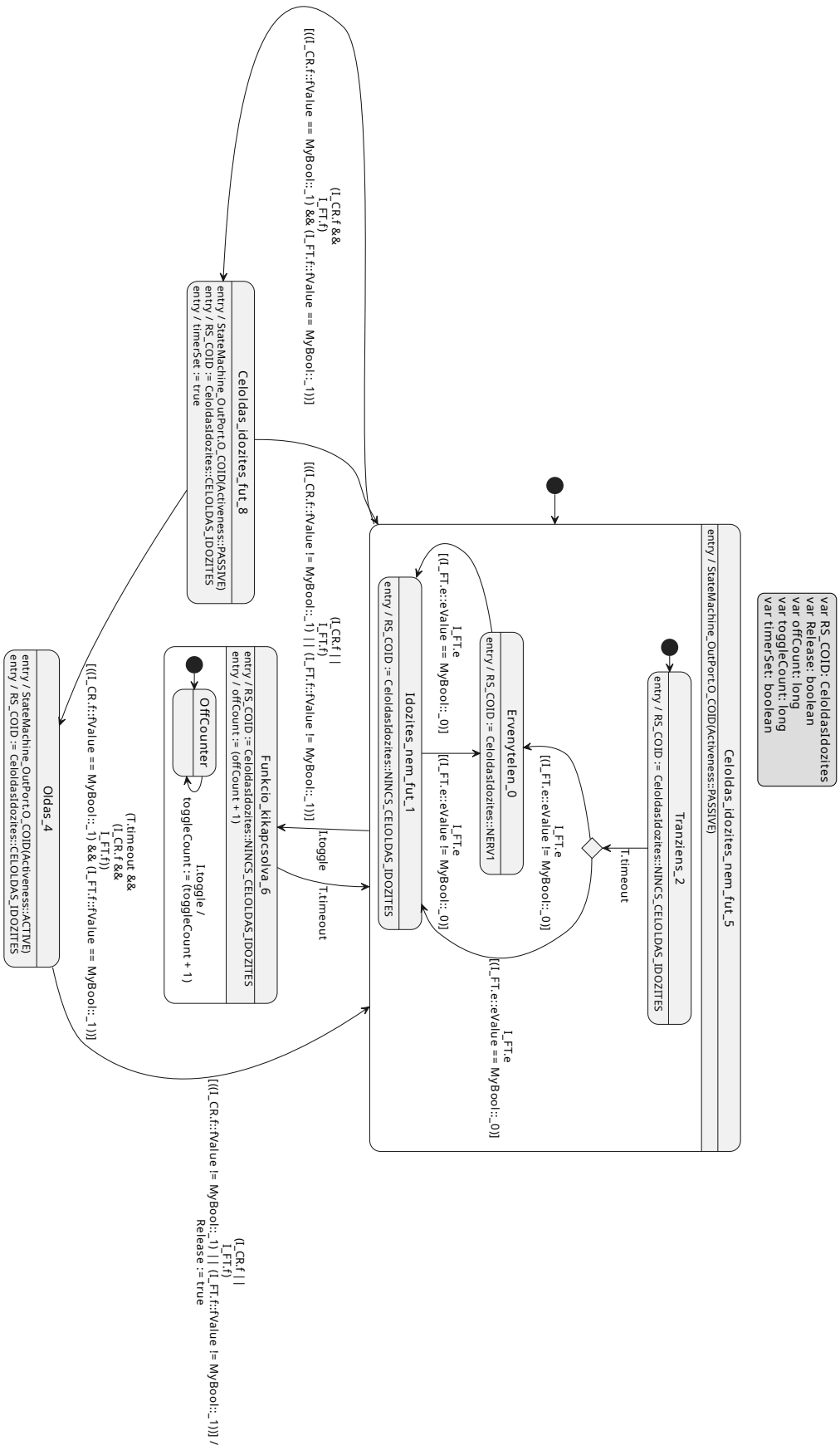



**Figure A.0.3:** Signaller model of the Railway Traffic Control System case study.