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

Extending the Capabilities of the CEGAR Model Checking Algorithm

Master's Thesis

Author
Zsófia Ádám

Advisor
Zoltán Micskei, PhD

June 6, 2023

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2023. június 6.

Ádám Zsófia
hallgató

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Background **4**

2.1 Verification of Critical Embedded Systems 4

2.2 Formal Verification and Model Checking 5

2.2.1 Abstraction in Model Checking 5

2.2.1.1 Abstract Domains 6

2.2.1.2 ARG 7

2.2.1.3 Traces 7

2.2.1.4 Pruning back the ARG 8

2.3 The Models throughout Model Checking in Practice 8

I Abstraction-based Trace Generation to Validate Semantics in Formal Verifiers **10**

3 Validating Semantics of Verifiers **11**

3.1 Formal Verification Process 11

3.2 Problem Statement 12

3.3 Challenges of Semantics in Model Transformation 13

3.3.1 Example of Ambiguous Semantics 13

3.4 An Approach to E2E Validation of the Verification Process 14

3.4.1 Another Use Case: Mitigating Modeling Mistakes 15

4 Abstraction-based Trace Generation Algorithm **16**

4.1 Prerequisites of the Trace Generation Algorithm 16

4.1.1 Abstraction Capabilities 16

4.2 Generating Traces without Abstraction . . . 17
4.2.1 Trace Generation without Abstraction Example . . . 17
4.3 Utilizing Abstraction . . . 18
4.3.1 Inappropriate Abstraction Level . . . 19
4.3.2 Trace Generation with Abstraction Example . . . 20
4.4 Analysis of the Proposed Algorithm . . . 22
4.4.1 Coverage Guarantees . . . 22
4.4.1.1 Coverage on the ARG level . . . 23
4.4.1.2 Typical Coverages for Engineering Models . . . 24
4.4.2 Usability and Feasibility for Validation . . . 26
4.4.2.1 Examples of Tools with the Necessary Prerequisites . . . 26

5 Evaluation 28
5.1 Prototype Implementation . . . 28
5.1.1 Gamma and Theta . . . 28
5.1.2 Process and Implementation . . . 28
5.1.2.1 High Level View of the Process . . . 29
5.1.2.2 Implementing Abstraction-based Trace Generation in Theta 30
5.1.2.3 XSTS Specific Additions . . . 30
5.2 Evaluation Design . . . 31
5.2.1 Research Questions . . . 31
5.2.2 Process and Goal of the Evaluation . . . 31
5.2.2.1 End-to-End Validation . . . 31
5.2.2.2 Real-World Models . . . 32
5.3 Designing a Validation Modeling Suite for Gamma . . . 32
5.3.1 Understanding Gamma Models and Traces . . . 33
5.4 Results of the Case Studies . . . 33
5.4.1 RQ1: Quantitative Analysis of the Models and Traces . . . 34
5.4.2 RQ2: Validation Findings . . . 34
5.4.2.1 Missing Default Values in XSTS . . . 36
5.4.2.2 Order of Operations inbetween Stable State Configurations 37
5.4.2.3 Limitation of Parallel Executions . . . 38
5.4.2.4 Visualizing Transitions Crossing Composite states with Orthogonal Regions . . . 40
5.4.3 RQ3: Traces of Real-World Models . . . 43
5.5 Discussion . . . 45

II Runtime Monitoring of Refinement Progress in CEGAR-based Model Checking **46**

6 Monitoring Refinement Progress in CEGAR **47**

6.1 Hardships in Model Checking 47
6.2 Problem Statement 48
6.2.1 Assumptions about the CEGAR loop 48
6.2.2 Refinement Progress Issues 49
6.3 Improved Detection and Mitigation 50
6.3.1 Detection 51
6.3.1.1 Analysis 52
6.3.2 Mitigation 53
6.3.2.1 Issues with Infeasible Traces 54

7 Comparison of Runtime Monitoring Techniques on Software Benchmarks **56**

7.1 Experiment Design 56
7.1.1 Implementation 56
7.1.2 Input Models 57
7.1.3 Research Questions 57
7.1.4 CEGAR Configurations 57
7.1.5 Execution Environment 58
7.2 Results 59
7.2.1 Data Preprocessing 59
7.2.2 RQ1 - Explicit Analysis 59
7.2.2.1 Detection for Explicit Value Analysis 60
7.2.2.2 Mitigation for Explicit Value Analysis 60
7.2.2.3 Differences in Execution Time 61
7.2.3 RQ2 - Predicate Analysis 62
7.2.4 RQ3 - Tracking ARGs 63
7.3 Conclusion 64
7.3.1 Threats to Validity 64

III Related Work and Conclusion **67**

8 Related Work **68**

8.1 The Landscape of Verification Tools 68
8.2 Test Generation with Model Checkers 69

8.3 V&V of Model Transformations . . . 69
8.4 Conformance Testing of Different Tools and Compilers . . . 69
8.5 Heuristics and Optimizations in CEGAR . . . 70

9 Conclusion **71**
9.1 Summary of Results . . . 71
9.2 Future Work . . . 71
9.2.1 Trace Generation . . . 71
9.2.2 Runtime Monitoring . . . 72

Bibliography **73**

Appendix **79**

Kivonat

A biztonságkritikus beágyazott rendszerek komplexitása folyamatosan növekszik, ideértve a szoftvereket és a mérnöki modelleket (pl. állapotgépek) is. Formális verifikációs eszközök segítségével bizonyos tulajdonságok teljesülése bizonyítható vagy hibák is megtalálhatóak, de ezek használata különböző kihívásokat rejt. Az ellenpélda alapú absztrakció-finomítás (CEGAR) egy jól konfigurálható formális verifikációs módszer, amelynek kiegészítése új technikákkal egy aktív kutatási terület.

A célom ebben a munkában a CEGAR kiterjesztése a formális verifikációs eszközök alkalmazhatóságának javítására. Két konkrét kihívásra összpontosítottam: I) az eszközök helyességével kapcsolatban a modelltranszformáció validálására, II) a finomítás hiányának detektálására futásidejű monitorozással a teljesítmény javítása érdekében.

1. rész Az első részben a mérnöki modellekre és azok formális verifikációs eszközökben való formalizálására összpontosítottam. A formális verifikációs eszközökben szükség van a szemantika formalizálására a bemeneti modell formális reprezentációra transzformálása során. Ez egy komplex feladat, mivel a szemantika sokszor alulspecifikált. Ezen transzformációk validációjára a jelenlegi gyakorlat teszt modellek készítése az adott modellezési nyelven majd érvényes lefutások megadása (általában manuálisan), majd ezek összehasonlítása a verifikációs eszköz által visszaadott lefutásokkal a konformancia bizonyítására. Ez a módszer hibákra hajlamos és nem hatékony, mivel bizonyos érvényes lefutások könnyen kihagyásra kerülhetnek.

Munkámban absztrakció-alapú modellellenőrzők 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.

Az algoritmust egy esettanulmány során ki is értékeltem reaktív állapotgépekhez készült eszközökön. Továbbá azt is ismertetem, az algoritmus milyen más felhasználásokra ad lehetőséget, például modellezési hibák azonosítására.

2. rész Visszatérve a verifikációhoz, a formális verifikáció és a modellellenőrzés tipikus kihívása a teljesítmény. Nagy bemeneti programok, például valós szoftverkomponensek, ellenőrzése nem egyszerű, mivel nincs olyan konfiguráció, amely minden adott bemenethez jól teljesítene. A terminálás jellemzően nem garantálható.

Korábbi munkáim során észrevettem a finomítás leállásának problémáját, amely megakadályozhatja a verifikációs algoritmus sikerét. Ennek megoldása érdekében futásidejű monitorozást javasoltam és ezt egy komplex portfólióban alkalmaztam. Jelen munkám során mélyebben elemeztem a problémát, részletezve az okait és hatásait, illetve frissítettem, javítottam és konfigurálhatóvá tettem a monitorozási technikáimat.

A kiértékelésben a futásidejű monitorozást különböző CEGAR és monitor konfigurációk alkalmazásával az SV-COMP de facto standard szoftverellenőrzési benchmarkjain futtatom, majd az eredményeket elemzem.

Abstract

Safety-critical systems are becoming increasingly complex, including both software and engineering models (e.g. state machines). Formal verifiers automatically prove properties or find errors in these models, but their usage hides various challenges. Counterexample-guided Abstraction Refinement (CEGAR) is a highly configurable formal verification metho. Adding new techniques to it for better performance is an active research area.

My goal in this work is extending CEGAR in ways to improve usability and applicability of formal verifiers. I focused on two specific challenges: I) correctness of the tools, mainly concentrating on validation of the model transformation, II) runtime monitoring of the formal verification, detecting the lack of refinement progress to improve performance.

Part I In the first part I focused on engineering models and how they are formalized in model checkers. Formal verification necessitates the precise definition of execution semantics of the engineering modeling language to be able to transform the engineering model to a formal representation. This is a complex task, as these semantics are usually underspecified. To validate these model transformations, the current state of practice recommends creating test models, defining valid execution traces for them (mostly manually), and comparing these traces to the ones returned by verifiers to check conformance. This is an inefficient and error-prone method as valid traces can be easily missed.

For abstraction-based model checkers I propose utilizing these abstraction-based techniques for automatic trace generation. I designed a trace generation algorithm, which is able to handle several cases with infinite state space. It utilizes abstraction in a configurable way and generates traces that do not unnecessarily repeat already covered states.

I evaluate the algorithm through a case study on tools for reactive state machines. I also show the possibility of other use cases, such as the mitigation of modeling mistakes.

Part II Returning to verification, a typical challenge for formal verification and model checking is performance. Large input programs, such as real-world software components, are not easy to verify, as there is no configuration that performs well for any given input. However, termination typically can not be guaranteed.

In my earlier work I noticed an issue of refinement progress halting, stopping the convergence of the verification algorithm to success. I proposed runtime monitoring to solve this issue and used it in a complex portfolio. In this work, I was able to assess the issue and my runtime monitoring techniques better, resolving the causes and effects of the issue in more detail and updating, correcting and making runtime monitoring configurable.

This time I evaluate runtime monitoring in detail by executing different CEGAR and monitor configurations on the de-facto standard software verification benchmarks of SV-COMP and analyzing the results.

Chapter 1

Introduction

With the increasing complexity of *safety critical systems*, *verification techniques* are experiencing a growing importance. Models play a central role in the design and development of these systems and thus the verification of not just software code, but also design models is a crucial step [53, 58]. 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 [34], proving the system safe for the given property. One of the best known formal verification techniques is *model checking* [6, 47].

Model checking [33] 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 [25], symbolic methods [27] or *abstraction-based techniques* [32, 33].

There are many different *model checking tools* in different application domains [54, 12], implementing these algorithms and enabling their users to automatically check different models (e.g. software code [12], hardware models [54] and so on). Different application domains come with different challenges, e.g. regarding soundness, correctness or performance.

Counterexample-guided Abstraction Refinement (CEGAR) is a highly configurable model checking method with many existing techniques for abstraction and refinement [49, 20, 70]. My goal in this work is extending CEGAR in ways to improve usability and applicability of formal verifiers. I focused on two specific challenges.

Part I correctness of model transformation in formal verifiers. The core part of these tools is the analysis itself on a formal model, but the correctness of the inevitable transformation step from the engineering model to the formal representation is just as important, however it is not trivial.

Termination for CEGAR executions typically can not be guaranteed. In practice, a time limit is set up and the execution is stopped with an inconclusive (timeout) result when this time limit is reached. In Part II I show how and why it is possible that a timeout is caused by an “infinite CEGAR-loop” due to lack of refinement progress and propose runtime monitoring to detect and mitigate such cases.

Part I: Trace Generation for Validating Semantics

Problem Statement Formal verification 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 [7].

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 [32], this is often not the case for the *model transformation* and optimization steps preceding the model checking algorithm.

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 [64].

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 [41], but it typically suffers from issues mainly caused by the blackbox usage of model checkers [40].

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 [61] and Theta [69], 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 [2]. 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.

Part II: Runtime Monitoring of Refinement Progress

Problem Statement Formal verifiers often face performance problems, hindering their applicability. Large input programs, such as real-world software components, are challenging to verify, as formal verification techniques are computationally expensive and there is no configuration that performs well for any given input.

Although the input model is often simply just too large and complex to be verified within a realistic time limit, this is not always the case. In my earlier work [1] I noticed an issue of refinement progress halting, stopping the convergence of the verification algorithm to success. In this work I analyze this issue in greater depth including causes and effects from an algorithmic standpoint.

Solution Proposal In my BSc thesis [1] I proposed runtime monitoring to solve this issue and used this technique in a complex portfolio, but it did not receive focused attention. In this work I revisit this monitoring technique and introduce several additions.

Detection and mitigation are split into separate components – both are heuristics in practice and other mitigation methods might also be possible, so they should be available separately.

Earlier several artifacts of the analysis were monitored, but it is not trivial if this is necessary or beneficial, so a configuration tracking only the counterexamples is also added, which will be used in the experiments during evaluation. Furthermore, mitigation included an issue discovered since then, which is also fixed in the current thesis.

Deeper analysis on when and how these techniques might be beneficial is also added separately both for detection and mitigation, which stem from empirically collected experiences since my BSc thesis.

Evaluation The runtime monitoring techniques were already used in a portfolio with many other techniques, but were not separately evaluated beforehand. The benchmarking experiment of this work concentrates on the different aspects of these techniques by running 9 different configurations on 4079 programs of the software benchmarking set of SV-COMP [12] and comparing how the different runtime monitoring techniques perform with different CEGAR configurations.

Summary Both proposed techniques of this thesis were successful when evaluated and show promising results in improving model checking for real-world use. Thus the objective of this work to improve CEGAR in practical use is achieved, while possibilities of further research on these topics is also opened.

Chapter 2

Background

This chapter provides the necessary background and context for this thesis: 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.

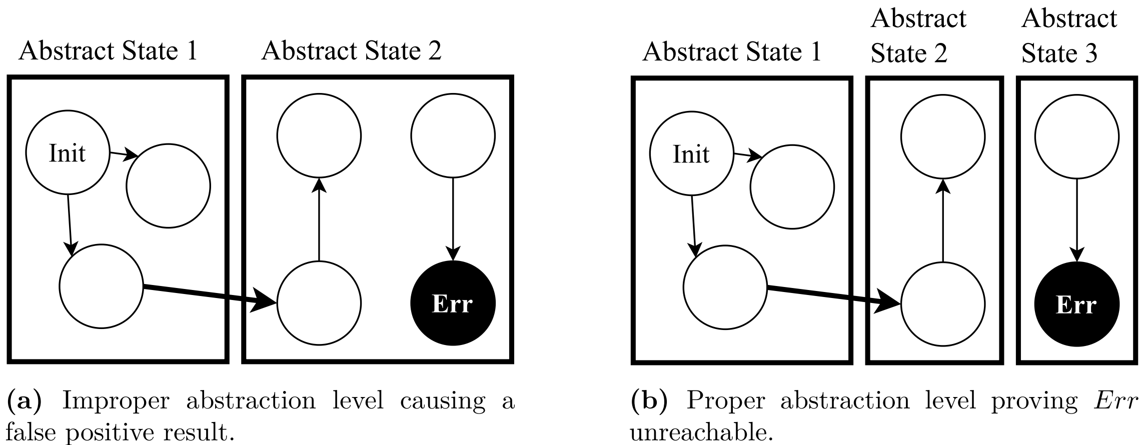
When models are extensively used during design, verification of such engineering models is a crucial part of these processes. Different project goals and types of models require different techniques, such as testing [44] or simulation. These are often complementary techniques used with different goals 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 [26, 48].

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 [52], or generated based on models (e.g. with the W or Wp method [26]).

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 [34] 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 [25], symbolic methods [27] or abstraction [32, 33].

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) [32], where abstraction and refinement are combined in a loop alternating the two as shown in Figure 2.2. Details of Figure 2.2 are detailed below.

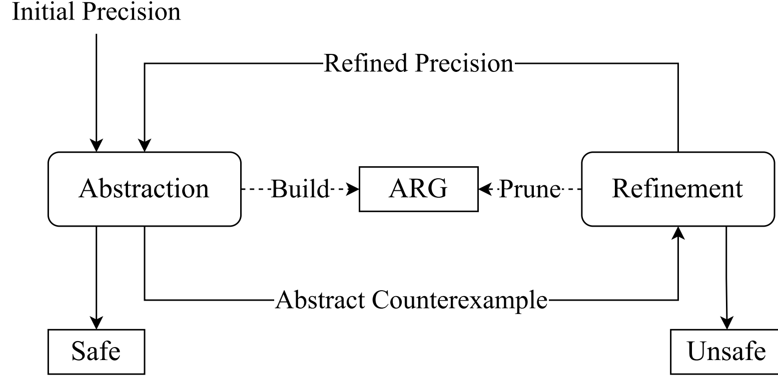


Figure 2.2: The CEGAR loop.

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 [18]). 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 [18]). Let π 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 π. ▪

There are many possible abstract domains, e.g. Cartesian predicate abstraction [45], boolean predicate abstraction [9], explicit-value abstraction [16] or even combinations of these and others [5]. Although the trace generation methods in this work will focus on the explicit-value domain, but predicate abstraction will also play an important role later on.

Explicit-value Abstraction [16] 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.

Predicate Abstraction [8] Cartesian and boolean predicate abstraction tracks a list of predicates as precision instead of variables. Cartesian abstraction keeps track of a conjunction of these predicates, while boolean predicate abstraction allows any arbitrary predicate combinations, not just conjunctions.

Expressive Power of Abstract Domains Most abstract domains have limitations in what they can express. These limitations cause a loss of precision, i.e. there is information that can not be expressed in that domain [8]. This can prevent the success of the analysis, if this information would be crucial to express.

Although the manner of limitations is diverse, expressive power can often be compared inbetween domains. For example, compared to explicit abstraction the Cartesian predicate domain is more expressive, e.g. both of these domains can track $x = 1$, but predicate abstraction can also add more complicated predicates, such as $x > 1$. However, Cartesian predicate domain tracks a conjunction of predicates, not arbitrary combination, which still introduces its own limitations.

Larger expressive power introduces its own advantages and disadvantages: it enables the model checker to be more precise, but also causes a significant performance overhead [8].

2.2.1.2 ARG

Abstraction based model checkers traverse an abstract state space building an *Abstract Reachability Graph* (ARG) [17].

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

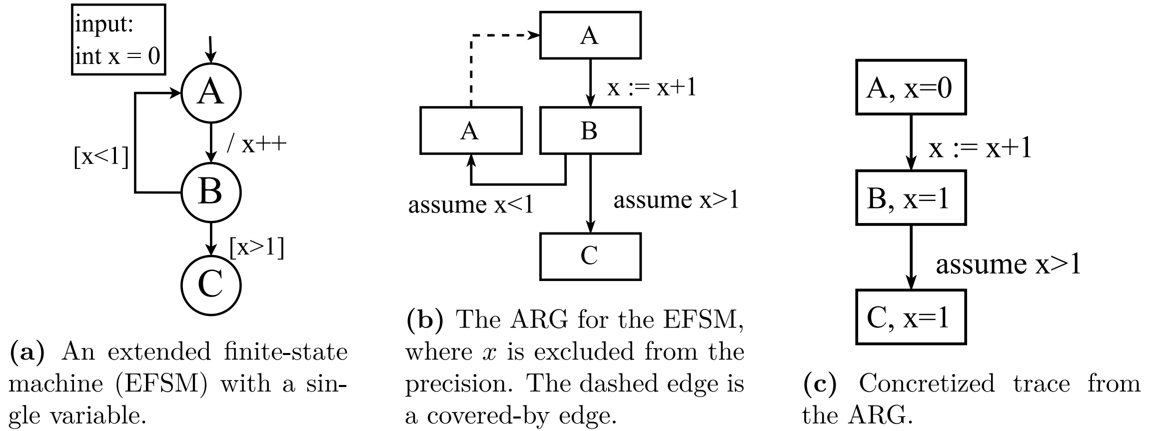
- $N \subseteq S$ is the set of *nodes*, each corresponding to an abstract state in some domain.
- $E \subseteq N \times N$ is the set of directed *edges*. An edge $(s_1, s_2) \in E$ is present s_2 is a successor of s_1 with regards to the transfer function T.
- $C \subseteq N \times N$ is the set of *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. The traversal can happen using many kinds of graph search algorithms from a simple BFS to complex heuristics.

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 2.3. The abstract states (rectangles) of the ARG only include the states of the EFSM, but not the value of x.



(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 2.3: Example of an ARG and concretized trace in the explicit domain.

There is an abstract trace in the ARG to C, which can be concretized to the trace shown in Figure 2.3c, 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.

2.2.1.4 Pruning back the ARG

Another point of configurability for refinement is how it modifies the ARG. The most trivial example is *full pruning*, where the ARG is pruned back to it's root and the abstraction algorithms will start building it using the new precision from scratch. Another example is what we call *lazy pruning*, which only prunes the counterexample under refinement back to the point where actual refinement occurred [49].

2.3 The Models throughout Model Checking 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 2.4 and explained in the paragraphs below.

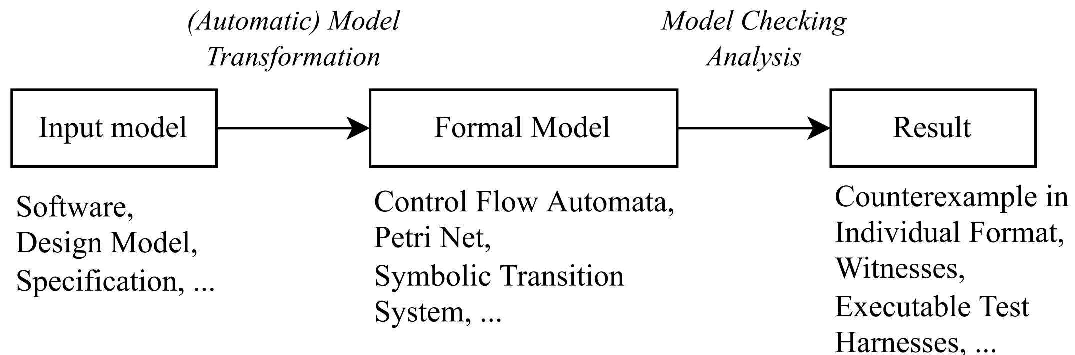


Figure 2.4: Typical models throughout the verification process.

Input Models Model checking is a widely used method with many different application domains, e.g. hardware specifications [66], software [12], protocols [30, 35], engineering and business models [54, 61]. Thus input models are often design models or software code instead of unambiguous formal models.

In this thesis 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) [7, 14].

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 [24], which runs the faulty execution.

There are also initiatives in software model checking for a uniform format, called witness [21, 22]. 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 thesis. Thus it is further elaborated on in Chapter 3.

Part I

Abstraction-based Trace Generation to Validate Semantics in Formal Verifiers

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 I.

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.

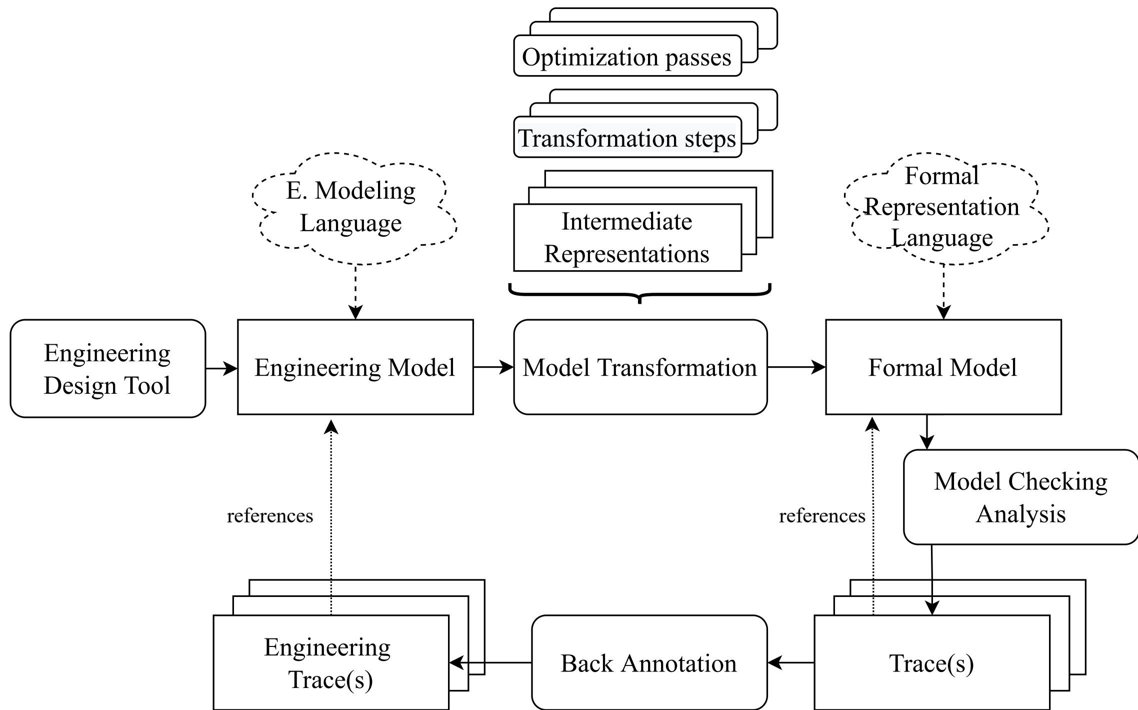


Figure 3.1: Typical process of a model checking toolchain.

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 [34]. A lot of work goes into the correct formalization of the algorithms used in the analysis and also to proving that they are correct [32].

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 [64, 65], *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 [42] 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.

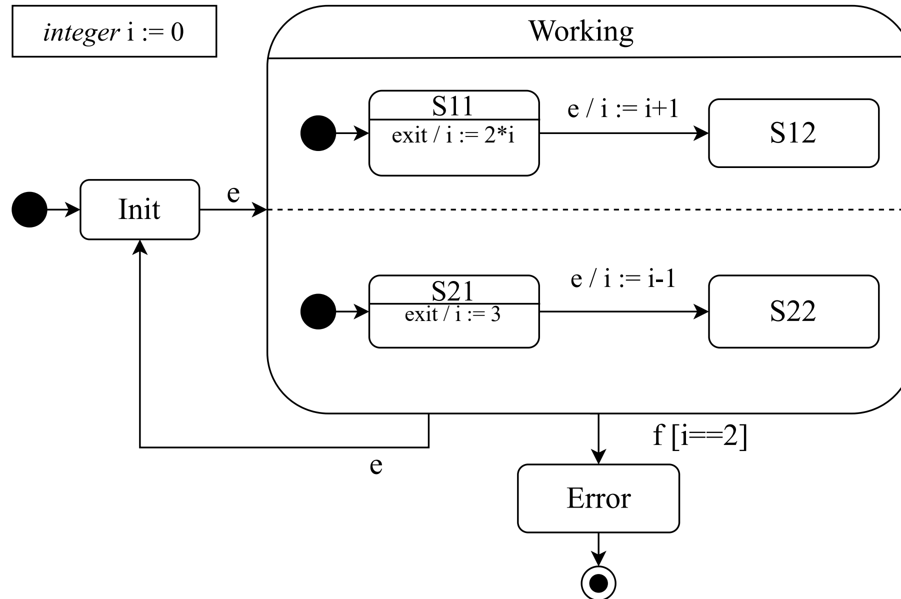


Figure 3.2: State machine containing language elements with ambiguous semantics.

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.

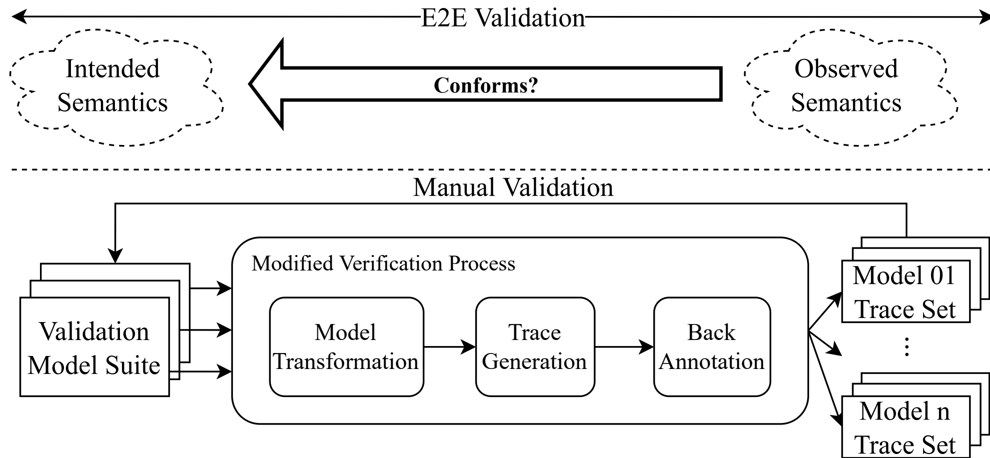


Figure 3.3: End-to-end (E2E) validation with trace generation.

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 [33] 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 [16]* (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 [17]* 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 [17] 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.

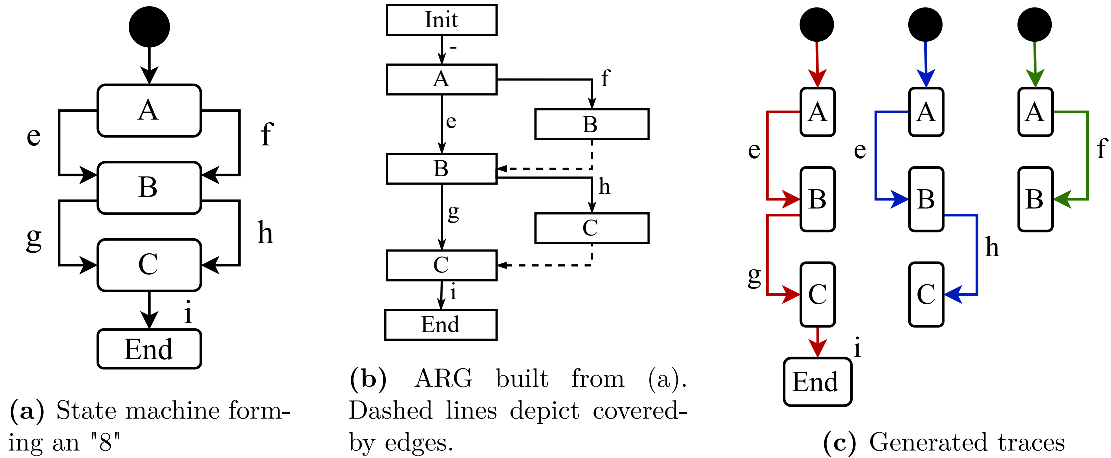
Algorithm 4.1: Trace generation algorithm without abstraction.

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

1. π: Initial precision created including every variable
2. $ARG(N, E, C)$:= `buildARG(`π`, `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.

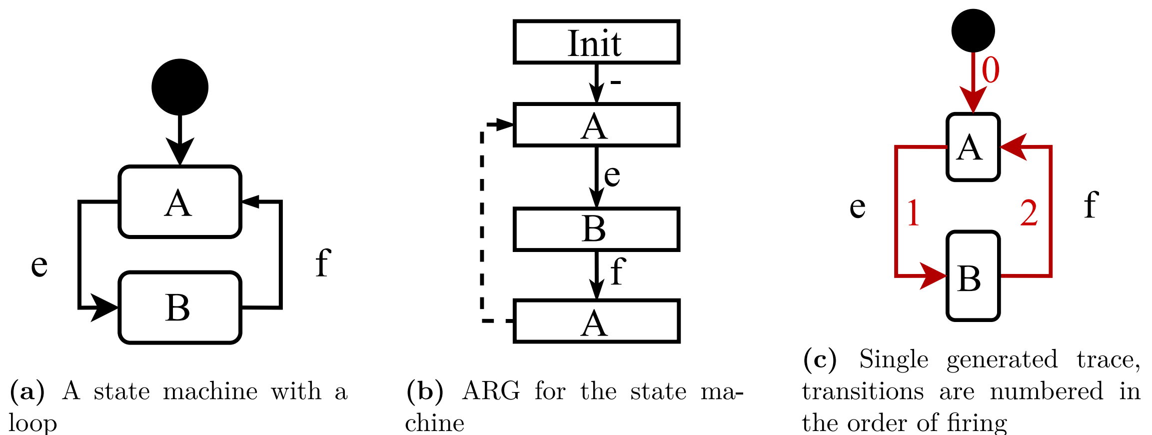


(a) State machine forming an "8"

(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

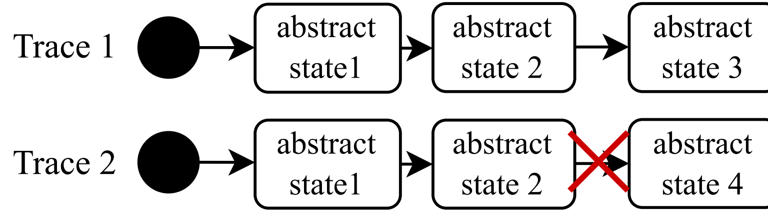


Figure 4.3: Trace 2 is only feasible if shortened, but then it is contained by Trace 1.

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 [34]).

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* [10] 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.

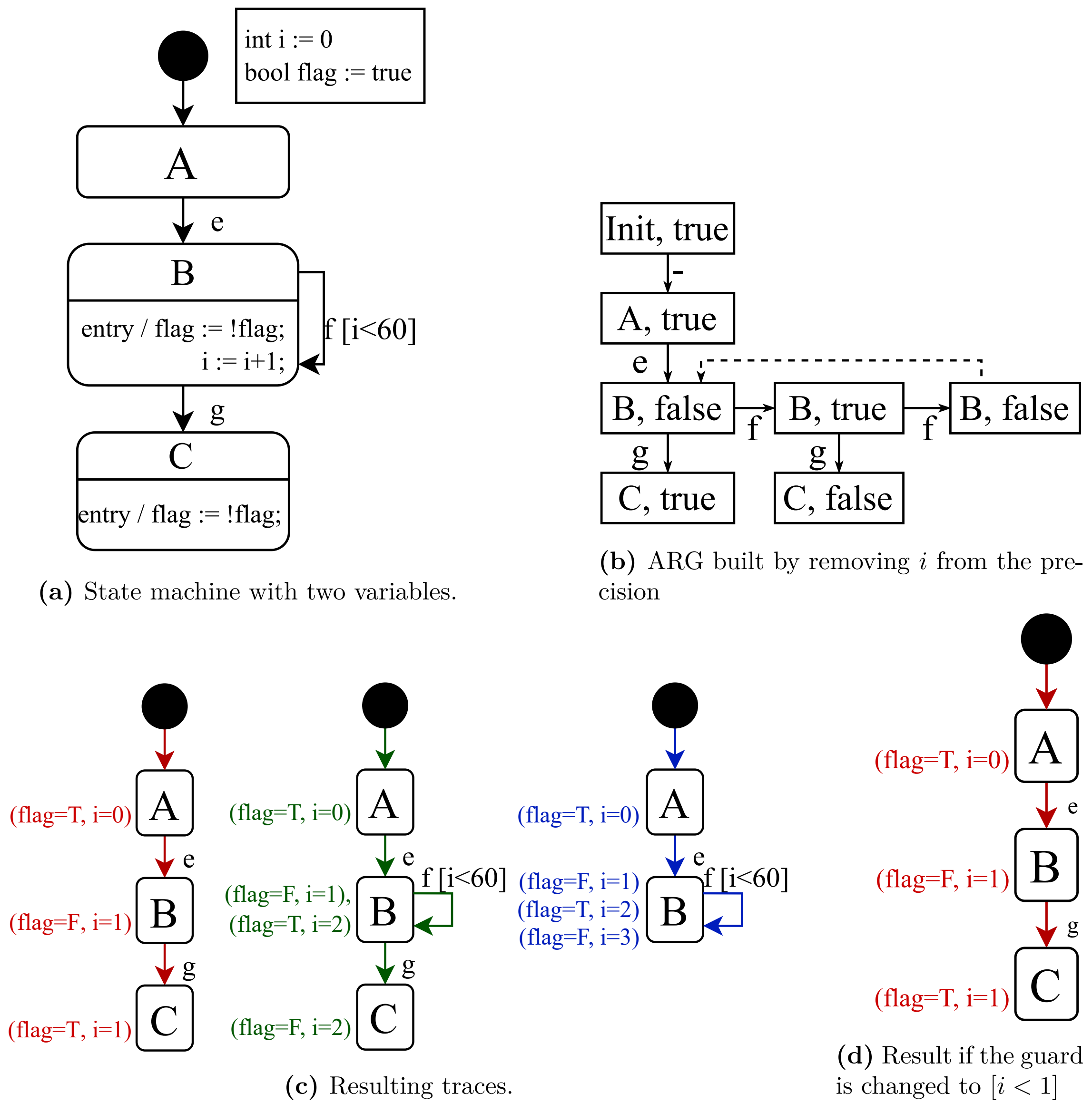
Algorithm 4.2: Trace Generation Algorithm with Abstraction.

input : F: Formal model, V: Set of variables to be included in the precision
output: T: Set of generated traces, $fullCoverage$: True, iff every abstract state is included in at least one trace

```
1  π: Initial precision including V
2  ARG(N, E, C) := buildARG(π, F)
3  n_0 : initial node of ARG
4  N_leaf := ∀n ∈ N where n has no outgoing edge
5  for ∀n ∈ N_leaf do
6  |   T ← trace from n_0 to n
7  T_concretizable := ∅
8  N_included := ∅
9  N_pruned := ∅
10 for ∀t in T do
11 |   if ¬ isFeasible(t) then
12 |   |   t′ := shorten(t)
13 |   |   N_pruned ← ∀n ∈ t, ∉ t′
14 |   if |t′| > 0 then
15 |   |   N_included ← ∀n node of t′
16 |   |   T_concretizable ← t′
17 for ∀t_a, t_b ∈ T_concretizable, |t_a| ≤ |t_b| do
18 |   if t_a starts with t_b then
19 |   |   T_concretizable := T_concretizable \ t_a
20 if ∃n ∈ N_pruned, ∉ N_included then
21 |   fullCoverage := False
22 else
23 |   fullCoverage := True
24 T_concrete := ∅
25 for ∀t ∈ T_concretizable do
26 |   T_concrete ← concretize(t)
27 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.

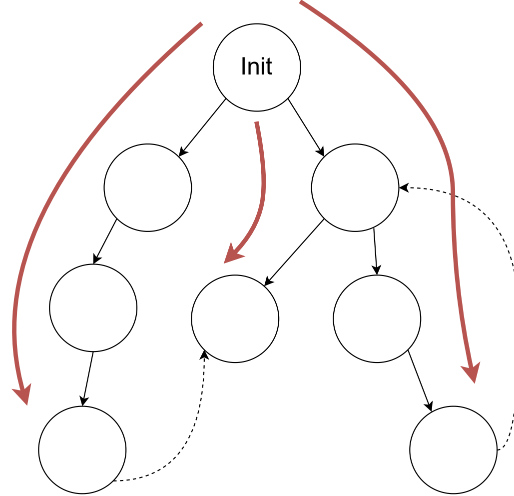


Figure 4.5: Possible ARG structure to illustrate ARG node coverage. The nodes are (abstract) states, the thick arrows show the abstract traces found in the ARG.

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 *(does not cover untracked variables)*
Possible Issues	State Space Explosion *(timeout)*	Inappropriate Abstraction Level *(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 [72, 74]). This criterion is satisfied iff each state of the state machine is visited. ▪

Definition 5 (All-Configurations Criterion [74]). 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 [31]). This criterion is satisfied iff each transition in the statechart is traversed. ▪

Definition 7 (Transition-Pair Criterion [63]). This criterion is satisfied iff for each pair of adjacent transitions exists a trace that traverses the transitions in sequence. ▪

Definition 8 (Decision Criterion [72, 74]). 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 [72, 74]). 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 [72, 74]). 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	Trace generation with Abstraction: No state space coverage violation detected	Trace generation with Abstraction: 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*) [68] 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 [34], 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 [61] and Theta [3, 69]. 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 [14, 36] 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 [67, 75] 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 [57] 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 [61] is an open-source modeling toolset with the goal of adding integrated verification and code generation features. It supports UPPAAL [55] and Theta [4, 69] 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 [4, 69] 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 [32], 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/

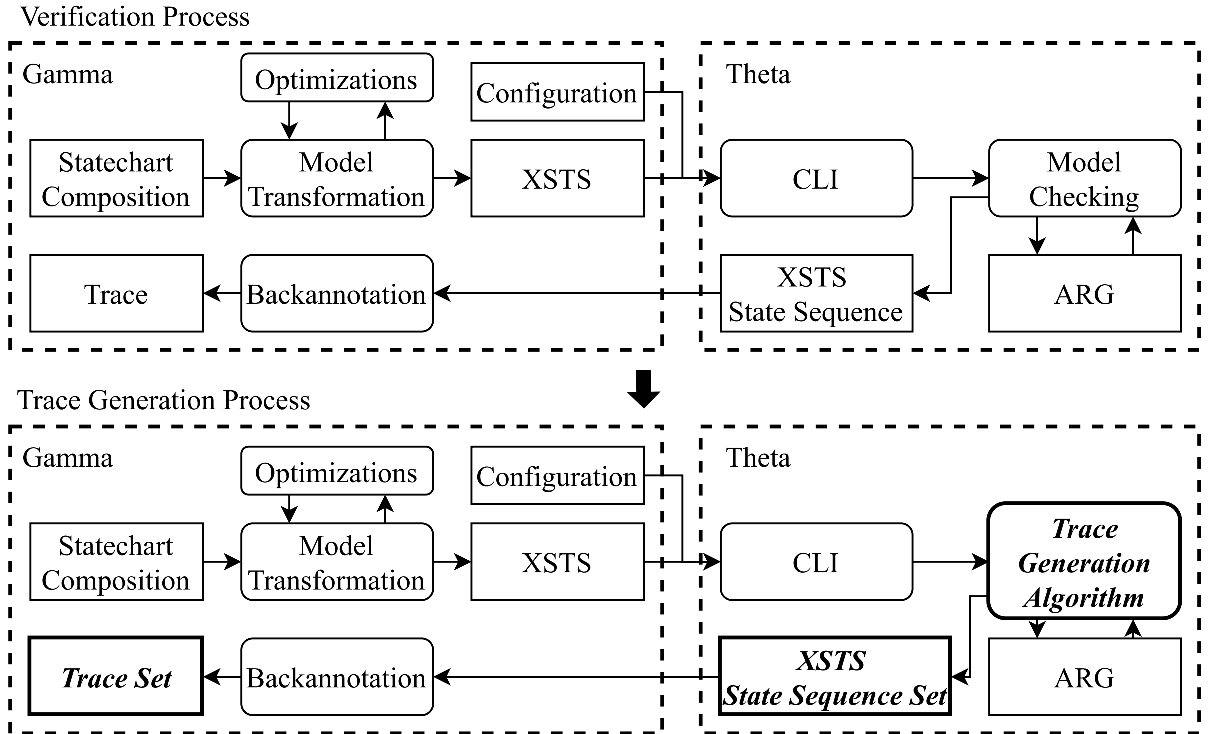


Figure 5.1: Verification and trace generation process

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) [62].

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) [62]. 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 part of the thesis, 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

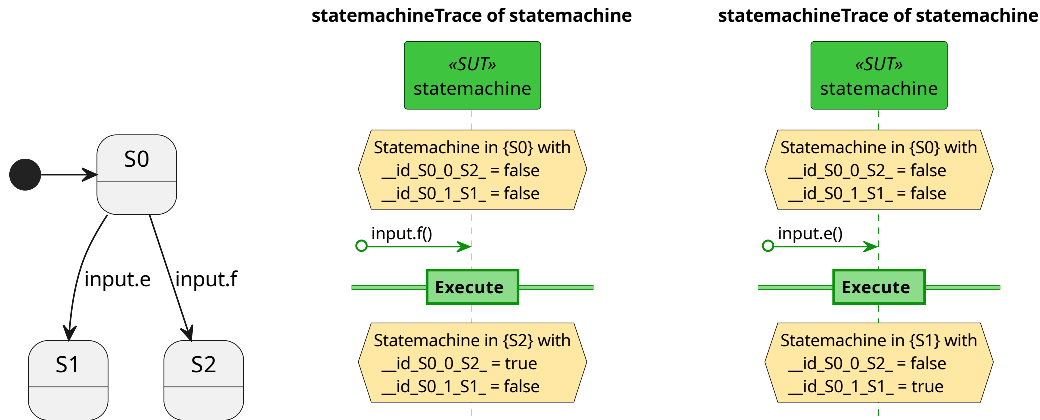
I systematically designed and modeled a validation modeling suite consisting of Gamma synchronous state machines for the E2E validation case study. The design followed iterative principles, i.e. it starts with a minimal set of model elements and progressively adds more and more elements to the models. 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 [61] 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 thesis, but are available as an artifact [2].

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).

Model (Package F)	St.	Tr.	Var.	Basic		Abstraction	
				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).

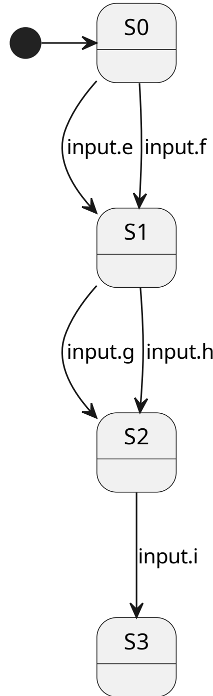


Figure 5.3: “Form 8” state machine modeled in Gamma (model 08, package A).

5.4.2.1 Missing Default Values in XSTS

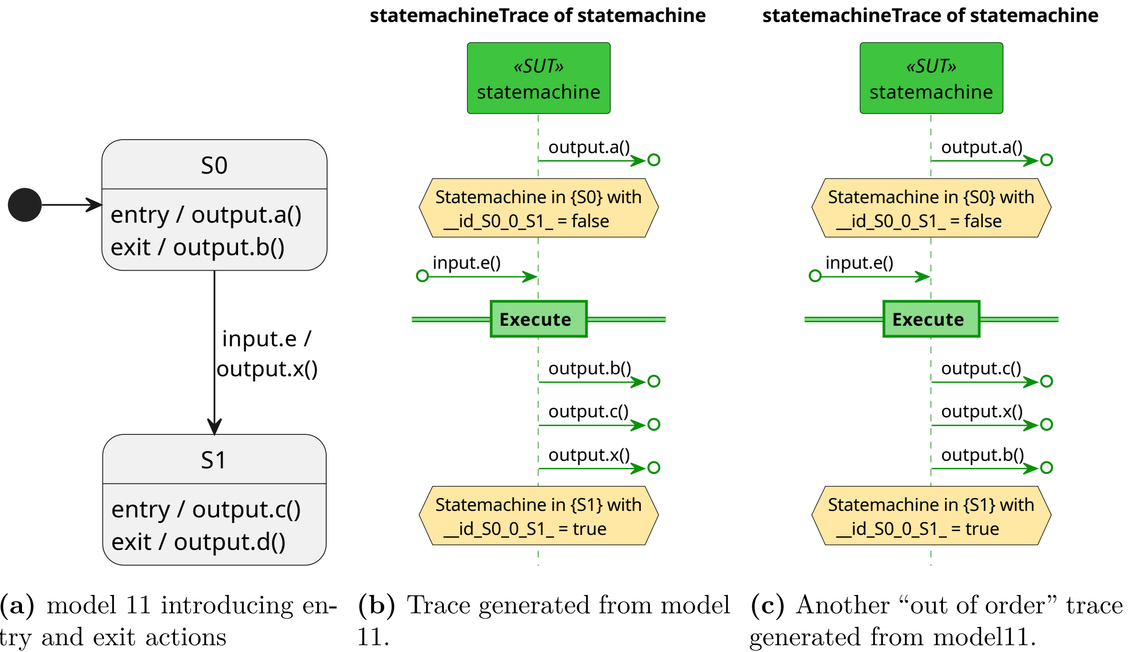
Discovery The first issue was found by discovering that a lot of traces are generated more than once when executing the algorithm. After checking the formal model and the ARG built for the model, a bug was found in the model transformation in Gamma.

Explanation As mentioned in Section 5.1.2.3, XSTS models [62] 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

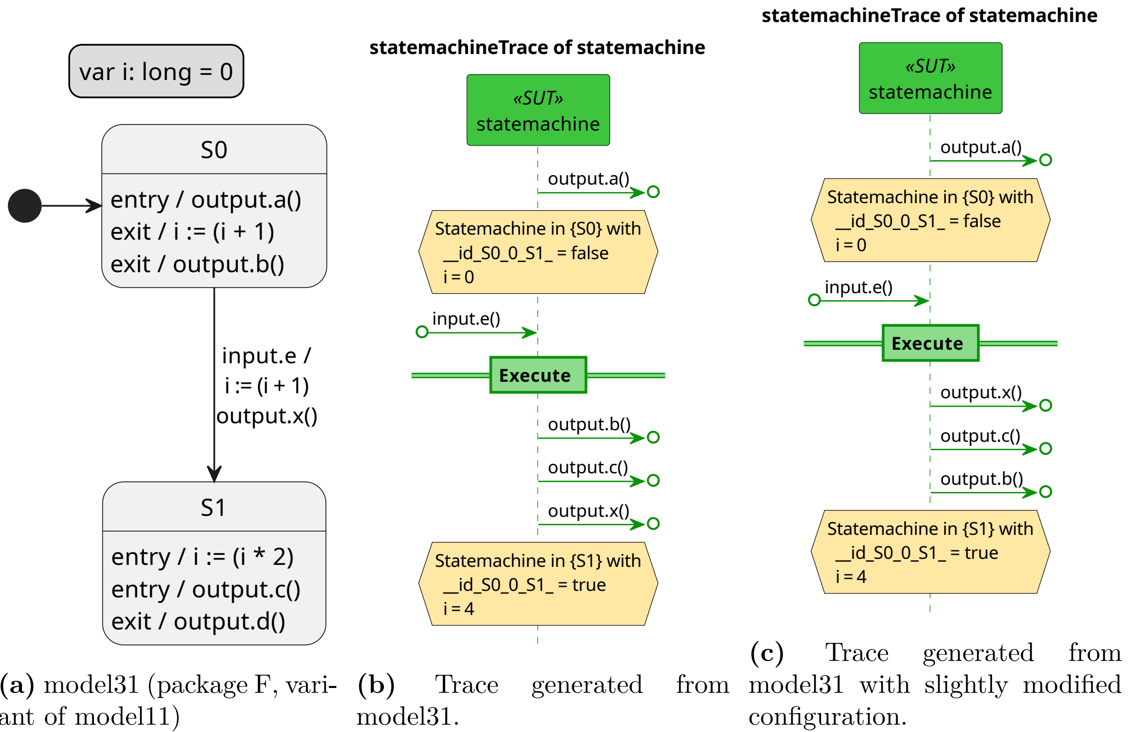
Discovery The naively expected semantics for the order of actions taken when a transition fires would be to execute the proper exit actions then the action on the transition and lastly the proper entry actions. Executing model 11 in package C it was discovered that the order of the output messages is 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 thesis 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

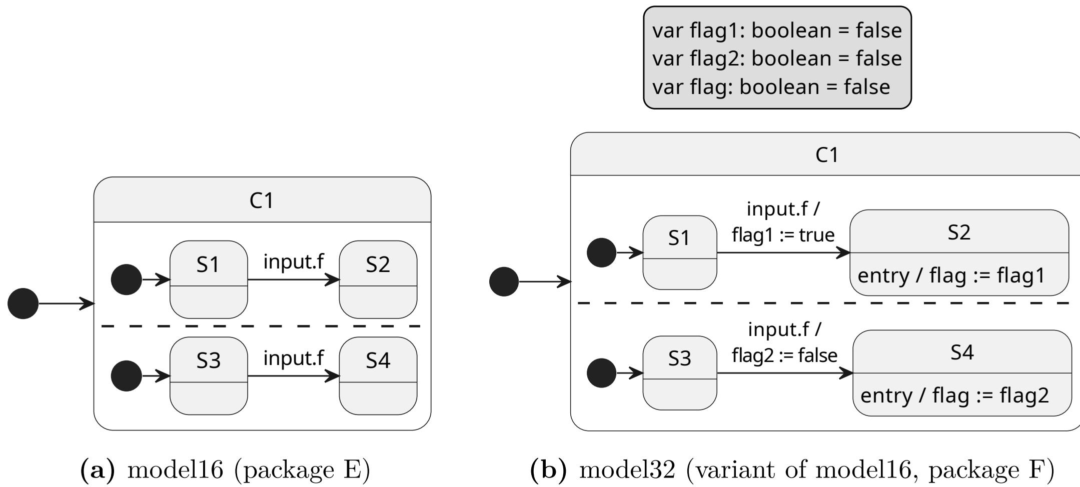
Discovery This finding stems from checking the traces generated by model16 and model32. The two models, shown in Figure 5.6, are variants of each other. Model16 belongs in package E, as it uses orthogonal regions, while model32 belongs in package F, as it extends model16 with variables.

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.

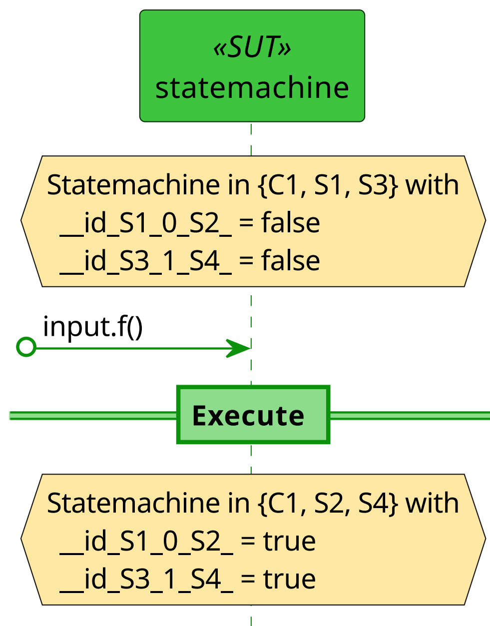
Explanation The semantical resolution for the issues in model32 in Gamma is that orthogonal regions are meant to be independent from each other. Gamma even issues a warning in the editor if instead of separate variables a single *flag* variable is used: “Both this transition and the transition between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

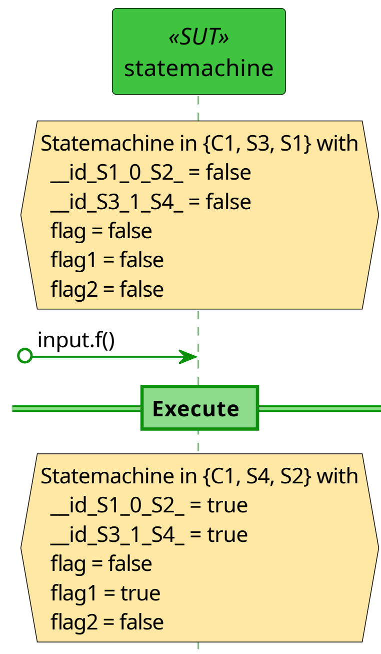


(a) model16 (package E) **(b)** model32 (variant of model16, package F)

Figure 5.6: model16 and model32, showing orthogonal regions

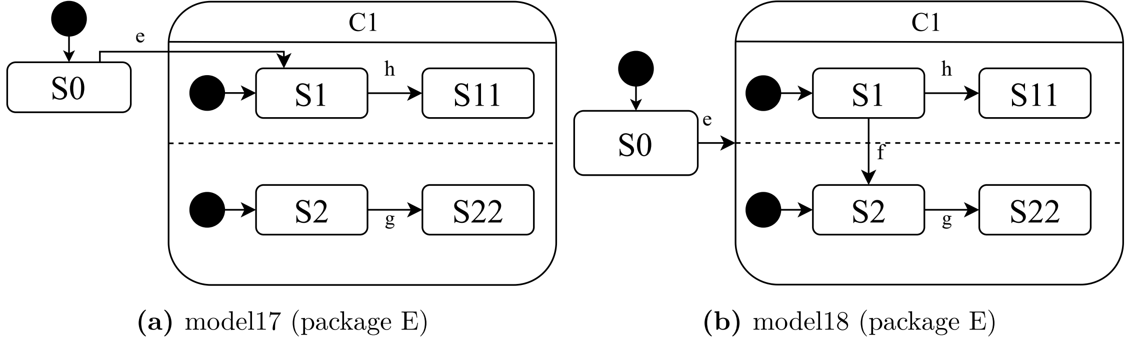


(a) Trace generated for model16.



(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.

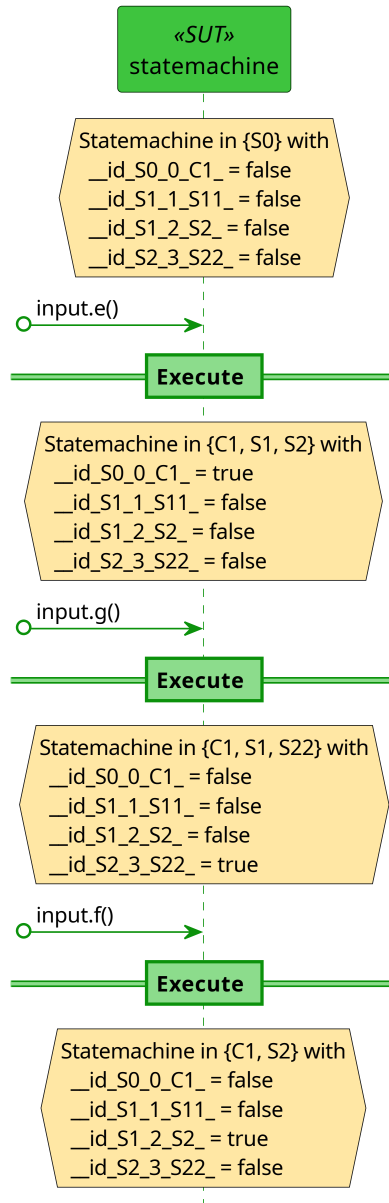
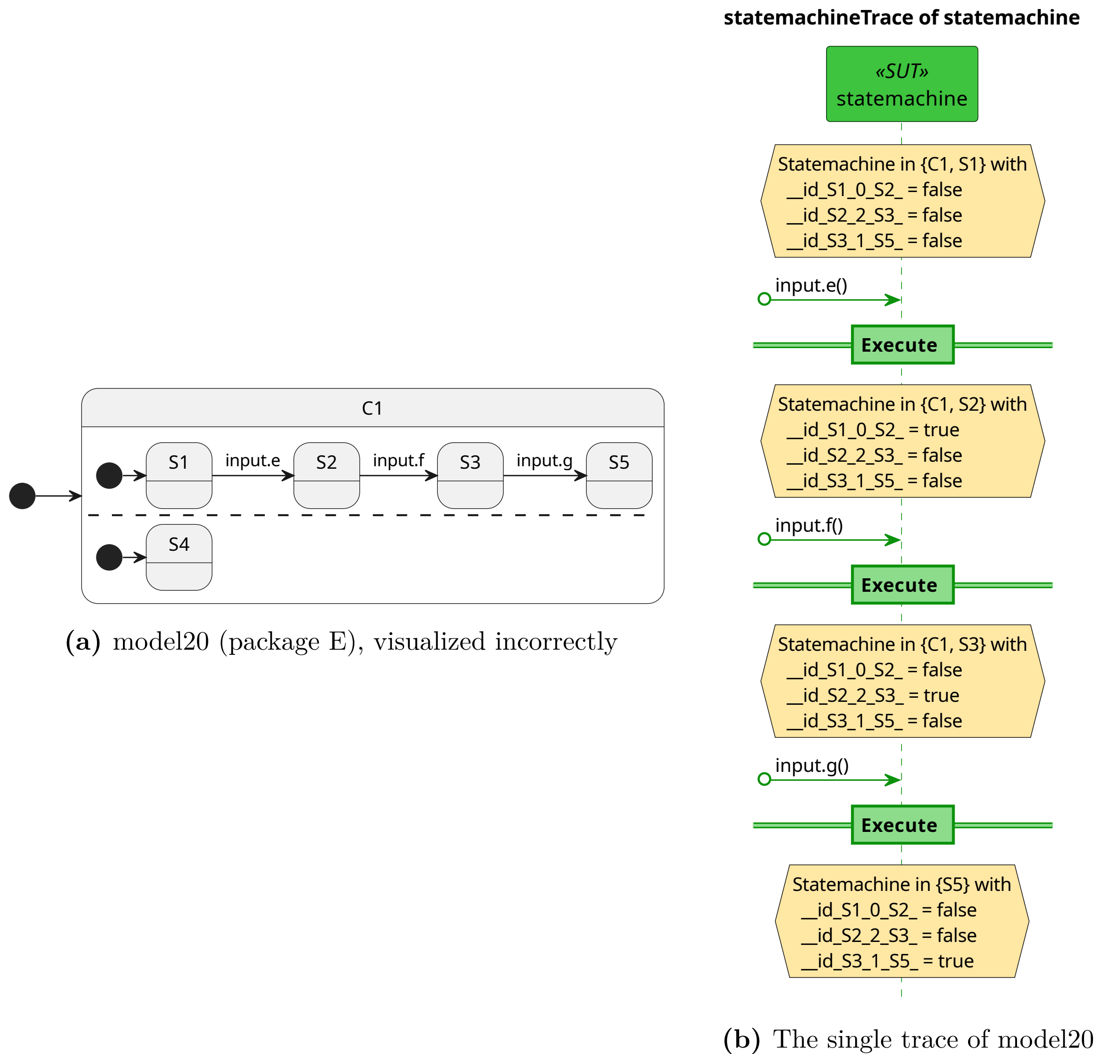


Figure 5.9: One of the traces for model18, enabling the model to reside in only one of the orthogonal regions



(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 [46] 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

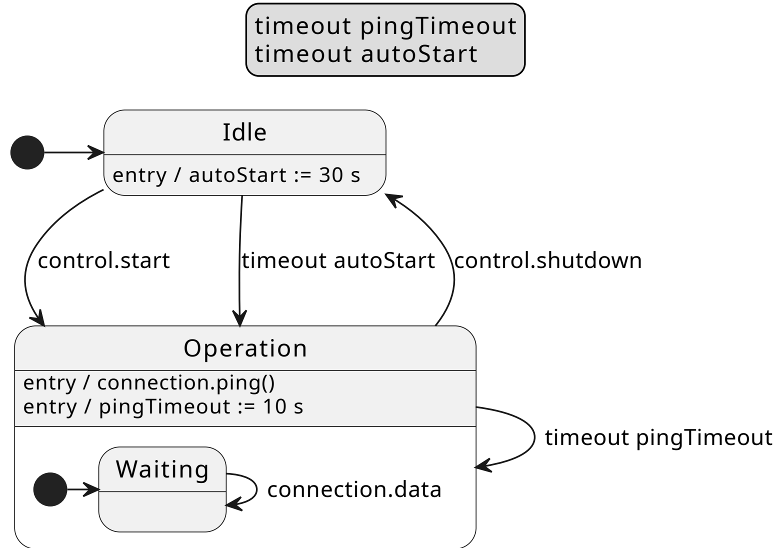


Figure 5.11: Ground Station model of the Simple Space Mission case study.

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.

Part II

Runtime Monitoring of Refinement Progress in CEGAR-based Model Checking

Chapter 6

Monitoring Refinement Progress in CEGAR

In my BSc thesis [1] I mainly focused on portfolios and algorithm selection for model checkers, including monitoring and intervention as means for a dynamic portfolio.

In Part II of this thesis I would like to revisit this earlier work in greater depth, this time concentrating not on portfolios, but refinement progress issues in CEGAR and the monitoring aspect itself.

First, I present an overview about issues in model checking in practice in Section 6.1. I was also able to dive deeper into how and why the issue of refinement progress halting is present in CEGAR, which I report on in Section 6.2. Next I revisited my earlier work about the methods of monitoring and mitigating this issue, updating these techniques and also giving a deeper analysis in Section 6.3.

6.1 Hardships in Model Checking

Ideally when we are using a model checker we would expect a (hopefully correct) answer, *safe* or *unsafe*, within a given time limit. In reality, this happy path is prone to not happening and instead we can get many other (unsuccessful) results:

- *timeout*, i.e. no result within the time limit,
- *out of resources*, e.g. mainly out of memory or stack,
- *different errors*, e.g. frontend issues, solver errors and so on.

A lot of the time the reasons for these results are trivial – e.g. more memory is necessary, the model needs to be changed so the tool can parse it, the solver implementation needs an update and so forth. *Timeouts* can often be trivial as well – we just need to give the tool more time (or computational power) and it will succeed. Hopefully "more" does not mean months or years.

However, this is not necessarily what's happening. Model checkers often face the challenge of tackling an NP-hard problem efficiently for as many real-world models, as possible [34]. Thus they can easily run into executions, where they will never terminate [71] and the user will have no idea about that.

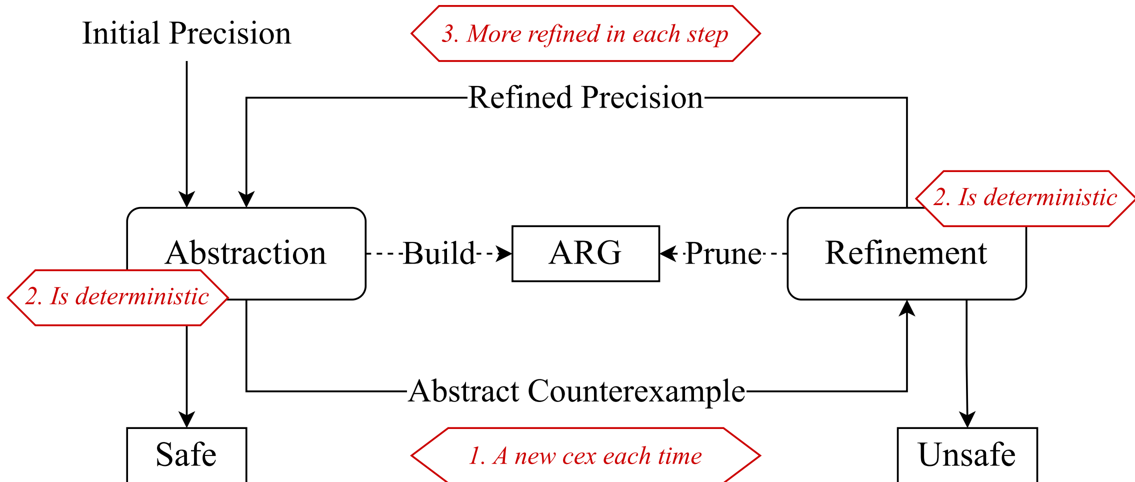


Figure 6.1: CEGAR loop with additional assumptions.

6.2 Problem Statement

From here on the main focus will be on CEGAR instead of model checking in general. The basic premise of CEGAR is finding the right level of abstraction to prove the system safe or unsafe, while the state space remains at a manageable size. Thus, we usually start with a really coarse over-approximation, iteratively refining the precision and thus the abstraction.

> However refinement progress can come to a halt due to information loss, which can prevent the CEGAR-loop's convergence to success.

Section 6.2.1 goes into detail about how the CEGAR loop is usually expected to behave, why this is often not the case and what issues this may cause. Then Section 6.2.2 focuses on the main issue of refinement progress stopping during the verification.

6.2.1 Assumptions about the CEGAR loop

The CEGAR loop in Figure 6.1 shows how the analysis should progress: the abstraction and refinement algorithms both work on the Abstract Reachability Graph (ARG), while also returning either an abstract counterexample or a refined precision to one another.

We are looking at the abstraction and refinement algorithms as black boxes – the CEGAR loop in this sense is mainly a framework and many different abstraction and refinement methods can be added to it, including a lot of heuristics [20, 37, 49].

However, we still tend make some general assumptions about this process – Figure 6.1 shows some of these assumptions (numbered 1-4). These assumptions are not stern rules that have to be obeyed each time – they are "idealistic" expectations and they will not hold for a lot of CEGAR configurations and heuristics. But if they are not complied with, then corner cases have to be thought through, otherwise surprising and unwanted results might arise.

1. Abstraction returns a new counterexample each time As stated above, the basic premise of CEGAR is the progression of refinement. Refinement is usually based

on the abstract counterexample found during the abstraction step. Thus it is sensible to expect different counterexamples each time for refinement to progress.

2. Abstraction and Refinement are deterministic If these algorithms are not deterministic throughout executions and iterations, different unwanted occurrences might arise, mainly that the model checker itself becomes non-deterministic, e.g. rerunning executions might not always result in success and thus benchmarking and debugging the tool becomes much more complicated. So these assumptions are often foundational for the usability of the analysis.

3. Precision more refined in each step Again, CEGAR has to find the right abstraction level, which requires this abstraction level to change. If the precision is not already right for a proof and it won't change, the CEGAR execution will not be successful.

Although this by itself only necessitates the expectation that the precision *will* change at some point, one still tends to assume that the precision should always change, as the algorithm is supposed to find some important information in each infeasible counterexample.

Again, it is important to highlight, that these are not rules and it might not always be possible or advantageous to comply with them. However, they introduce the mindset of the problems and solutions given in the next sections.

6.2.2 Refinement Progress Issues

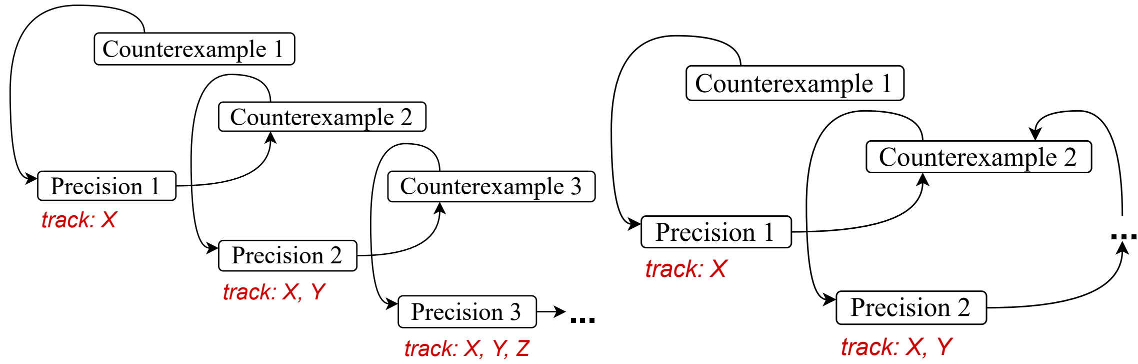
If the abstraction level of the precision can not progress during the analysis, i.e. the precision is not changed during the refinement step of the CEGAR loop, it can cause the progress of the analysis to come to a halt and get stuck in an "infinite CEGAR-loop".

To illustrate how this can happen, the assumptions from Figure 6.1 will be used.

- Assumption 2 expects the inner algorithms to be deterministic. This should be possible to implement, so we can assume that without a technical error or an explicit design choice this is generally true.
- Assumption 1 and 3 are connected in the sense that if one is compromised at some point, then the other can also get compromised from the next iteration onwards. However, these assumptions are not trivial to guarantee, so in this section we will examine the causes and consequences of that.

The determinism of the algorithms means that for the same input they should always produce the same output. An "opposite" of that would be to expect that for any two different inputs, the algorithms always return different outputs. However, this is not desirable, e.g. in explicit abstraction, if two different counterexamples are infeasible due to the same variable, then we want to add that same variable to the precision as refinement.

Assumption 1 and 3 realize something similar, but more relaxed and beneficial: in the iterative process of the CEGAR loop, where the precision is iteratively refined, the algorithms should give different and progressing outputs to the inputs they get. Figure 6.2a illustrates this: the refinement iteratively adds new variables (X, Y, Z) to the precision based on the last counterexample, which leads the abstraction to new counterexamples.



(a) Ideally the iterations of the CEGAR loop progress towards the right abstraction level.

(b) Instead of progress, execution might run into an infinite CEGAR loop.

Figure 6.2: Demonstration of ideal and problematic CEGAR executions with explicit abstraction.

This progression sounds reasonable to expect, but progress is not always possible in practice. The most common reason for that is the limited expressive power of abstract domains (see Section 2.2.1.1) and information loss due to different heuristics.

Example of No Progress The typical example for that is the explicit domain: the precision contains variables and the value these variables take up is tracked. However, the value of a variable can be unknown (i.e. it could hold more than one value). Thus, in those states it does not matter that we added it to the precision or not.

Figure 6.2b shows the main issue with this: we add Y as it plays an important role in showing that counterexample 2 is infeasible, but if we can not track its values in some states, we might end up on the same counterexample over and over and the progression of the CEGAR loop can come to a halt. Keep in mind, that for more complex configurations, e.g. with lazy pruning of the ARG, this infinite loop might be several counterexamples long.

Most other abstract domains, such as Cartesian predicate abstraction, might have greater expressive power, but they can still suffer from this issue. Although other reasons might also be possible, expressive power seems to be by far the most common culprit.

In the next section I would like to introduce a technique which tries to tackle this issue with runtime monitoring.

6.3 Improved Detection and Mitigation

In my BSc thesis I realized a technique of detecting and possibly mitigating this issue with runtime monitoring [1], which seemed promising, especially for the explicit domain. Since then I revisited these techniques, deconstructing my earlier work and uncovering possible issues, thus a formalization of the updated technique is due and is given in this section.

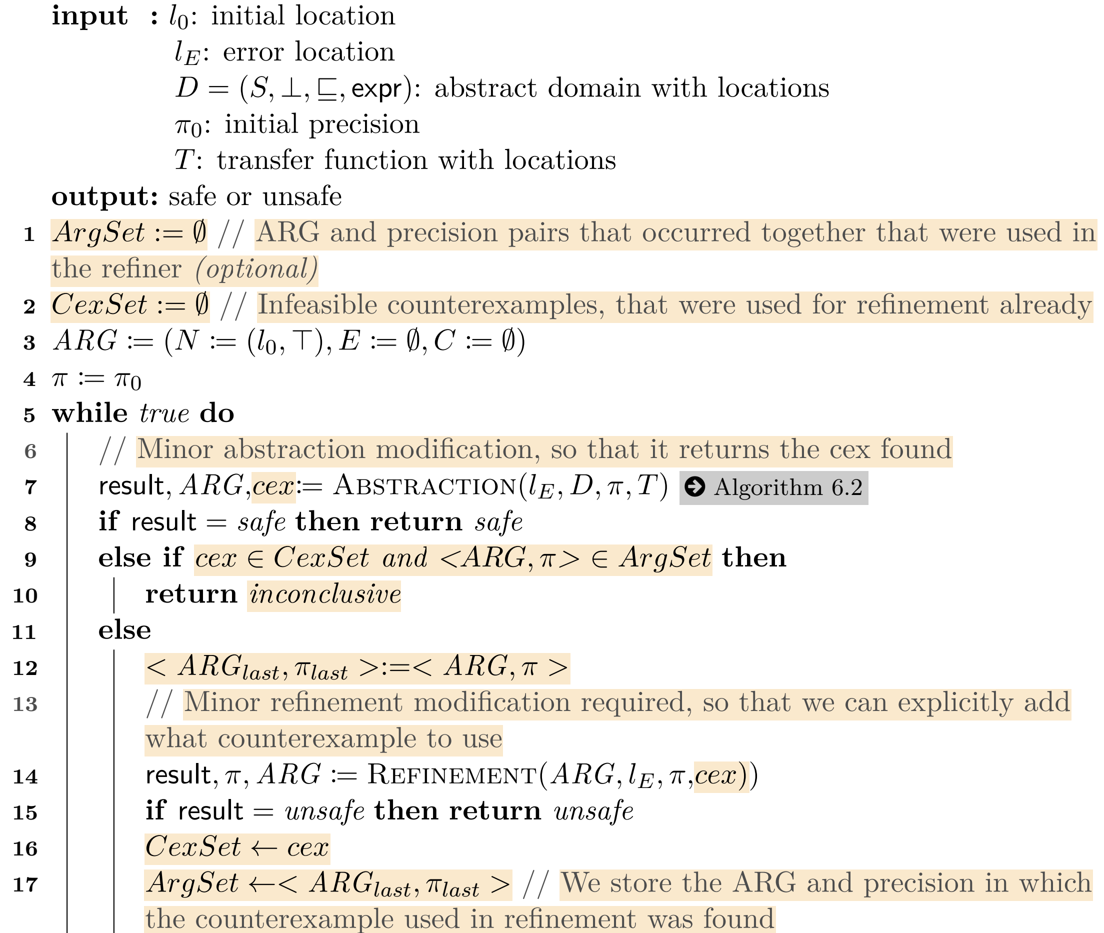
In this work I separated the runtime monitoring method to different, separately usable components, making it configurable, which will be a crucial part for the analysis below and in the evaluation as well:

- detection and mitigation are now separate components, as different mitigation methods might be preferred to the one introduced in this chapter,
- and tracking only the counterexamples or also the ARGs and precisions will also be separated.

Below I introduce the resulting components and detail the causes and results of this deconstruction.

6.3.1 Detection

The current version of the monitoring algorithm is formalised in Algorithm 6.1. It is added to the CEGAR loop and requires only a minor modification of parameters for the abstraction and refinement, which might be available in the implementation already. The parts realizing the detection of halting refinement progress are highlighted in orange.



Algorithm 6.1: Refinement Progress Detection in the CEGAR loop.

input : l_0: initial location
l_E: error location
$D = (S, \bot, \sqsubseteq, \mathsf{expr})$: abstract domain with locations
π_0: initial precision
T: transfer function with locations

output: safe or unsafe

1 $ArgSet := \emptyset$ // ARG and precision pairs that occurred together that were used in the refiner *(optional)*
2 $CexSet := \emptyset$ // Infeasible counterexamples, that were used for refinement already
3 $ARG := (N := (l_0, \top), E := \emptyset, C := \emptyset)$
4 $\pi := \pi_0$
5 **while** *true* **do**
6 // Minor abstraction modification, so that it returns the cex found
7 result, ARG,cex:= ABSTRACTION(l_E, D, π, T) ➔ Algorithm 6.2
8 **if** result $=$ *safe* **then return** *safe*
9 **else if** $cex \in CexSet$ *and* $<ARG, \pi> \in ArgSet$ **then**
10 **return** *inconclusive*
11 **else**
12 $< ARG_{last}, \pi_{last} >:=< ARG, \pi >$
13 // Minor refinement modification required, so that we can explicitly add what counterexample to use
14 result, π, ARG := REFINEMENT(ARG, l_E, π,cex))
15 **if** result $=$ *unsafe* **then return** *unsafe*
16 $CexSet \leftarrow cex$
17 $ArgSet \leftarrow< ARG_{last}, \pi_{last} >$ // We store the ARG and precision in which the counterexample used in refinement was found

The monitoring method presented in Algorithm 6.1 works the following way:

- An empty *CexSet* and *ArgSet* is initialized, which will store the infeasible counterexamples that were already used for refinement earlier (in practice we store the hashes of these structures).

- When abstraction returns with an abstract counterexample, we check if the tracked structures are already present in the sets, i.e. if we used them for refinement before; if they are, the analysis is stopped and deemed inconclusive, as we will not be able to progress with refinement in this case.
- If the counterexample and other structures are "new", we proceed to use it for refinement and if refinement deems it infeasible, we add the counterexample to the *CexSet* and the ARG and precision in/with which it was found to the *ArgSet* afterwards.

Some minor parameter modifications of abstraction and refinement are made: the explicit output or input of the abstract counterexample is required. This, however, should be easy to do in practice, if not already done. These algorithms do not need to be modified in any other way for the detection component.

6.3.1.1 Analysis

Improving Verification Results It is important to see that this extension of the CEGAR loop by itself will not be able to raise the number of successful analyses. However, it can cut down on the number of timeouts, returning with an inconclusive result much earlier on. This can already save immense amounts of time, but if it is is paired with any kind of mitigation heuristic, either inside the analysis (e.g. the technique below) or outside the analysis (e.g. a sequential portfolio of different analyses), the number of successfully verified input models can also rise.

Tracking different Structures In Algorithm 6.1 abstract counterexamples, ARGs and precisions are all stored and tracked. However, based on Section 6.2, tracking counterexamples by themselves should be enough to recognize if we are in an infinite CEGAR loop or not.

Yet in practice some CEGAR configurations might be able to mitigate this issue sometimes, e.g. if lazy pruning is used, the abstraction might be able to "pull itself out" from this situation in several iterations, as shown in Figure 6.3.

In Figure 6.3 we assume that the precision does not change throughout the three iterations. The red counterexample on the left is found first, but it is infeasible and pruned back the whole way. It is then rebuilt again in the next iteration, but the other branch of the ARG also gains a new state. The right counterexample is found after iterations of no precision or counterexample change, because it is slowly able to build up parallel to the wrong counterexample.

Thus the monitoring technique can be deemed a heuristic technique, in which we have to define at what point we would like to deem the CEGAR execution hopeless and inconclusive. Some examples are listed below.

- Tracking only counterexamples is an over-approximation for lazy abstraction with the BFS search strategy, i.e. it might produce inconclusive results, where lazy pruning could have "mended" the lack of refinement progress in a few iterations.
- On the other hand, tracking counterexamples only might be enough for lazy abstraction with a deterministic DFS search strategy, i.e. in that case lazy abstraction will not be able to mitigate itself the same way as with BFS.

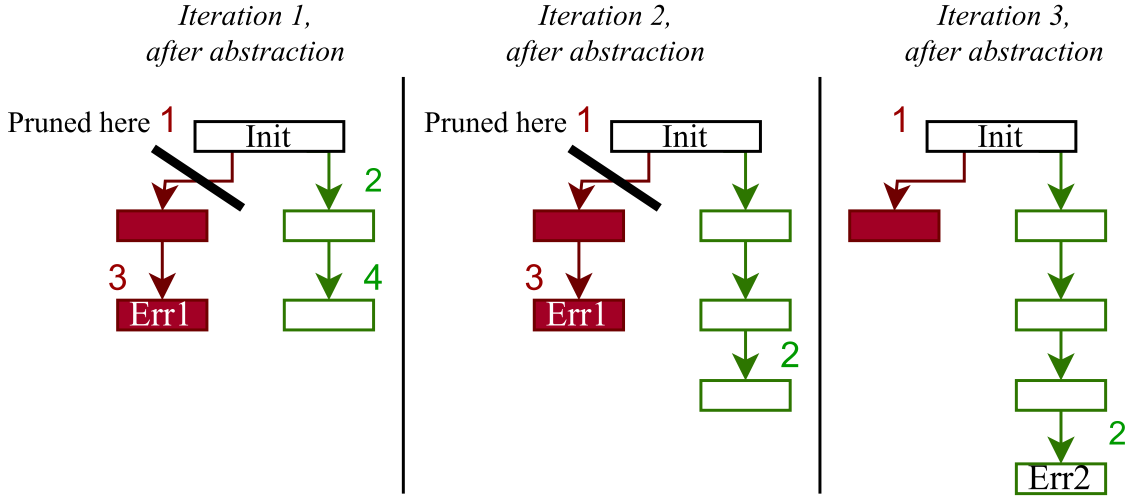


Figure 6.3: Illustration of how lazy pruning with BFS might be able to make progress even if the counterexample and precision remain the same. The numbers on the arrows show in what order the states are found.

- Also, if the "self-healing" situation of Figure 6.3 is generally deemed slow and rare, tracking counterexamples only and "false detection" of these situations can actually be worth it for the faster detection of no refinement progress in other cases.

Thus tracking only the counterexamples might be an over-approximation, but it might easily be the better option in practice, especially if paired with mitigation or other configurations. New configurations, which require the tracking of even more structures might also be possible, although in this work I will only discuss and evaluate the above two options.

6.3.2 Mitigation

In my earlier work I designed a fairly simple mitigation technique, but it was not separated from detection. It can be paired with other techniques, e.g. portfolios or it might be omitted entirely in favour of other techniques, as it is now completely separated from the detection component. However, the version in my earlier work also had a hidden issue, which will be detailed and fixed in Section 6.3.2.1.

This mitigation heuristic modifies the abstraction algorithm. The core idea is the modification of the stop criterion of ARG building so that instead of stopping at the first counterexample, the algorithm stops at the first "new" counterexample.

Algorithm 6.2 shows the modifications with the darker, green highlights, while the lighter highlights are the modifications that were already necessary for the detection component. The set of counterexamples (*CexSet*) and optional ARGs and precisions (*ArgSet*) are already familiar from Section 6.3.1. They are necessary for the mitigation as well, so they become part of the input for abstraction. When finding an error location and thus a counterexample, the sets are used to check if we have found a new counterexample or not.

This time finding a known counterexample will result in the continuation of the ARG building instead of stopping the analysis with an inconclusive result, preventing the halt of refinement progress by finding a new ARG.

Algorithm 6.2: Abstraction algorithm with refinement progress mitigation.

input : $ARG = (N, N_{notCovering}, E, C)$: partially constructed abstract reachability graph
l_E: error location
$D_L = (S_L, \bot_L, \sqsubseteq_L, \mathsf{expr}_L)$: abstract domain with locations
π_L: current precision
T_L: transfer function with locations
$CexSet$: A set of infeasible abstract counterexamples that were already found and refined earlier.
$ArgSet$: ARG and precision pairs that occurred together with counterexamples that were used in the refiner *(optional)*

output: (safe or unsafe, ARG, cex)

1 waitlist := unmarked nodes from N
2 **while** waitlist $\neq \emptyset$ **do**
3 $\quad l, s :=$ remove from waitlist
4 $\quad cex = ((l_1, s_1), op_1, \ldots, op_{n-1}, (l_n, s_n)) :=$ path to unsafe node (with l_E)
5 $\quad$ **if** $l = l_E$ *and not (cex ∈ CexSet and <ARG, π> ∈ ArgSet)* **then**
6 $\quad\quad$ **return** *(unsafe, ARG, cex)*
7 $\quad$ **else if** $l = l_E$ **then**
8 $\quad\quad$ // Remove covered-by edges
9 $\quad\quad$ **for** $\forall (l_i, s_i) \in cex$ **do**
10 $\quad\quad\quad$ **if** $\exists (l', s') \in \text{arg} : \{(l', s'), (l_i, s_i)\} \in C$ **then**
11 $\quad\quad\quad\quad C := C \setminus \{(l', s'), (l_i, s_i)\}$
12 $\quad\quad\quad\quad N_{notCovering} \leftarrow (l_i, s_i)$
13 $\quad$ coverOrExpand($ARG, l, s, T,$ waitlist$, D_L, \pi_L$) // not detailed, usual CEGAR steps
14 **if** $\exists (l_E, s) \in N$ **then**
15 $\quad$ **return** *(inconclusive, ARG, None)*
16 **else**
17 $\quad$ **return** *(safe, ARG, None)*

Finding a new counterexample might not always be possible, i.e. when the waitlist runs out without finding a new counterexample. With some further proofs of soundness it might be possible to conclude *safety* in that case, but the current version of this method will either conclude *unsafe* if finding a feasible counterexample *(in refinement)*, or *inconclusive (line 15)*. For this to work, line 9 and 10 of Algorithm 6.1 were moved to the end.

6.3.2.1 Issues with Infeasible Traces

The parts of the mitigation algorithm described above did not change much since my earlier work. However, line 7-12 of algorithm 6.2 were not explained yet.

Usually abstraction stops at the first counterexample. Even if that is not true, infeasible counterexamples, or at least some postfix of them (see lazy pruning [49]), are pruned back before the next iteration of ARG building.

However, this mitigation technique does not comply with the assumption that there are no infeasible traces in the ARG when it is built: ARG building does not necessarily stop at the first counterexample and only one counterexample will be refined and pruned, leaving

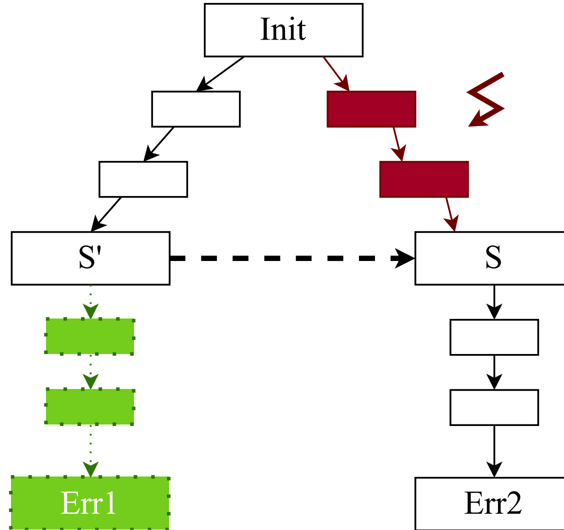


Figure 6.4: The issue with covered-by edges going from infeasible traces.

infeasible traces in the ARG behind, which might cause issues later on or even in that iteration, as the infeasible trace can cover other ARG states outside that infeasible trace.

An example of such an issue is shown in Figure 6.4. The figure shows and oversimplified ARG without labels that contains two traces. The infeasible counterexample on the right was found first – it's infeasibility stems from the ARG states before S contradicting each other. However, we know from earlier that this trace is infeasible and using the mitigation technique build the ARG further on. Then we find an S' state on another route, but this state can be covered by S, so instead of expanding it further a covered-by edge is added and the feasible counterexample to Err1 is never found and the model might even be considered safe in the end, which is false.

Keep in mind, that lazy pruning is not necessary for this example to play out, i.e. even if the ARG is built up from the initial node in each iteration, the analysis might have rebuilt the same counterexample again in lack of refinement progress.

The root of the issue was that the ARG by itself is not prepared to handle infeasible counterexamples remaining in it. The simplest solution is adding the possibility to "mark" known infeasible traces and preventing coverage of other states by the states in these traces.

In algorithm 6.2 line 7-12 implements this extension by maintaining $N_{notCovering}$, adding these marked nodes to it and also removing the already existing covered-by edges to them. Nodes added to $N_{notCovering}$ will be forbidden from covering other nodes. In practice this might instead be an attribute of the ARG nodes instead of a separate set in the ARG, but both achieves the same result.

What Mitigation to Use Again, it is important to see that this mitigation technique is a heuristic technique and will not always achieve better results than without it, e.g. if we use explicit analysis in a model with loops, it might prevent refinement progress from stopping, but it will not help with a large amount of infeasible counterexamples due to the loops. In that case using another abstract domain instead of waiting for the explicit analysis might be a better idea.

Chapter 7 will show how detection and mitigation perform in practice with different configurations.

Chapter 7

Comparison of Runtime Monitoring Techniques on Software Benchmarks

This chapter conducts an experiment defined to evaluate the runtime monitoring techniques introduced in Chapter 7. Section 7.1 details the design of this experiment from research questions and configurations used to implementation details and the execution environment. Then in Section 7.2 the results of the verification executions are evaluated based on the research questions introduced beforehand. Besides concluding this chapter Section 7.3 also accounts for the threats to validity in this evaluation.

7.1 Experiment Design

The following sections introduce the necessary details about experiment design.

The subject of the experiment are the different configurations of the monitoring techniques introduced in Chapter 6, evaluated through their implementation in Theta [69].

7.1.1 Implementation

The runtime monitoring techniques and the changes made to them in this work are implemented in the XCFA CLI tool of the Theta [69] model checking framework. The implementation is open source and available on github[1]. Different configurations can be set by using the `--cex-monitor` option and setting it to one of the following values: `DISABLE`, `CHECK`, `CHECK_ARG`, `MITIGATE`, `MITIGATE_ARG`. This will enable the separate benchmarking of all the components shown in Chapter 6.

The monitoring configurations implement the following: `DISABLE` serves as the baseline without any of my additional techniques, `CHECK` and `CHECK_ARG` only implement detection, but not mitigation and track only the counterexamples or also the ARGs and precisions, respectively. Additionally, `MITIGATE` and `MITIGATE_ARG` also implement the mitigation technique introduced in Chapter 6.

[1] https://github.com/ftsrg/theta/tree/progress-check-refactor

7.1.2 Input Models

In this experiment the input models for verification will be the benchmarking programs of the International Software Verification Competition (SV-COMP) [12]. SV-COMP provides the de-facto standard and probably the biggest benchmarking set for C verifiers.

In this experiment I will use the ReachSafety category, which provides C programs with a reachability property. The configuration files also contain if the programs are believed to be safe or unsafe, thus false results will also show. The exact version used can be found on gitlab[2].

This benchmarking set contains *6417* tasks (programs), of which Theta is able to parse *4097* programs (thus the rest will be omitted). These include different subcategories concentrating on different language elements of C, such as bitvectors, loops, arrays, floats and so on. The variation in size of the program is also large: some of the programs are small, i.e. easily readable, even less than 50 lines, some are really large, i.e. more than 10 000 lines.

7.1.3 Research Questions

RQ1: How well do the detection and mitigation techniques perform if using an abstract domain with low expressive power? The domain to used here will be the explicit domain, which I expect to be prone to the refinement progress issues. As the detection and mitigation components are now separate, they can be separately evaluated. As detailed in Section 6.3.2, it is not trivial if it is worth letting the analysis run further with mitigation or if it might be more advantageous to start another CEGAR configuration right away instead.

RQ2: How uncommon it is for refinement progress to halt if using an abstract domain with a large expressive power? Although the Cartesian predicate domain has a much larger expressive power, it is still possible to run into the refinement progress issue. The main interest here if the number of executions with this issue is negligible or not.

RQ3: What differences appear inbetween results of tracking only the counterexamples and tracking the ARGs and precisions as well Tracking only the counterexamples is a simpler solution, which is much easier to implement without errors and might be easier to extend with new techniques. However, when only tracking counterexamples, false positives can come up, especially with using configurations like lazy pruning, which might be able to "mend" itself (see Section 6.3.1.1).

7.1.4 CEGAR Configurations

For this evaluation I chose 3 different CEGAR configurations, each of which is executed with at least 2 of the 5 possible monitoring options, resulting in 9 combinations. The option "mitigate_arg" was however not used in this evaluation, as it was not necessary for answering the proposed questions.

[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/ecc7d14aa0715c3f51146c3ffa05048ce5e60be7

		Predicate Analysis	Explicit Value Analysis	
			Full Pruning	**Lazy Pruning**
CEGAR options	**Abstract Domain**	Cartesian Predicate	Explicit	Explicit
	Pruning Strategy	Full	Full	Lazy
	Refinement	Backwards Binary Interpolation		
	Search Strategy	ERR	ERR	BFS
Factors	**Runtime Monitoring**	*disable, check*	*disable, check, check_arg, mitigate*	*disable, check, check_arg*
	Relevant RQs	RQ2	RQ1, RQ3	RQ3

Table 7.1: Summary of the 9 combinations executed for this experiment. Each column shows a CEGAR configuration, while the rows show the chosen CEGAR and monitoring options. Throughout the experiment each CEGAR configuration was executed with the values in the runtime monitoring row for comparison.

The 3 chosen CEGAR configurations all use a generally well performing refinement technique and either error location based search, or in case of lazy pruning, BFS (as explained in Section 6.3.1) [49]. Beside these they only differ in the abstract domain and/or the pruning strategy as shown in Table 7.1.

7.1.5 Execution Environment

The experiment was executed on the cloud infrastructure of the Faculty of Electrical Engineering and Informatics. The load was distributed inbetween around 70 instances of the same virtual machine template, running Ubuntu 22.04 with 3 CPU cores and 15GB of RAM.

Execution of the tasks was conducted by BenchExec [23], the open-source, reliable benchmarking framework developed for and used at SV-COMP. In this framework the memory limit was set to 15GB of RAM (the maximum available amount), while the time limit was set to 500 seconds of CPU time.

Using virtual machines for the experiment was not ideal, but necessary. Some amount of deviation in time is expected even when not executing the tool on virtual machines as well, as Theta is implemented in Java and the garbage collector is executed in a non-deterministic manner. In general, we experienced no large deviations in results when re-running smaller parts of the experiment.

But the executions for some tasks with the same combinations might still output different results due to deviation in time, e.g. the same task might be solved once in 490 seconds, but it might not be solved under 500s the next time (but would be solved in a few more seconds).

Thus all 9 combinations were executed twice and any task, which had different results inbetween the two executions was removed from the benchmark set.

7.2 Results

In this section the results of the experiment are detailed by going through each research question and examining relevant data.

The resulting dataset is available freely on Zenodo [76] along with the executable and the Jupyter Notebook used to process and plot the results.

7.2.1 Data Preprocessing

Some preprocessing was necessary to analyze the resulting data:

- results of different technical errors (solver, generic, etc.) were merged into a single "ERROR" status, as the exact type of errors are out of scope for this experiment,
- as described in 7.1.5, the combinations were run twice and input tasks with different results inbetween the two runs in any of the combinations were removed everywhere *(235 tasks were removed in this step)*,
- right now there is a single task that Theta gives the wrong result to (false negative), which is a known, but for now unsolved issue, so this task was removed *(1 task was removed in this step).*

Due to these steps, the analysis below was carried out on executions on 3843 input programs for each combination.

Abbreviations of output in Table 7.2 and all similar tables mean the following:

ERR Technical Error (e.g. solver error)

OOM Out of Memory

T/O Timeout

False Success, input is unsafe

True Success, input is safe

Stuck Stopped due detection of lack of refinement progress

7.2.2 RQ1 - Explicit Analysis

To answer RQ1 we will compare results from 3 runtime monitoring options combined with the explicit value analysis with full pruning. The number of different results is shown in Table 7.2.

- Compared to the baseline, the number of timeouts is much lower if only detection is used and somewhat lower, if mitigation is used.
- With the exception of a few input tasks (which are present due to technical errors in Theta), the number of successful (false or true) results is the same with and without detection and higher if mitigation is used.

output	disable	check	mitigate
Err	967	901	1007
OOM	4	4	4
T/O	2460	1748	2310
False	194	194	285
True	218	217	217
Stuck	0	779	18

Table 7.2: Overview of the explicit analysis with full pruning results using no monitoring/detection/detection and mitigation.

		disable				
		ERR	**OOM**	**T/O**	**False**	**True**
check	**ERR**	901	0	0	0	0
	OOM	0	4	0	0	0
	T/O	0	0	1748	0	0
	False	0	0	0	194	0
	True	0	0	0	0	217
	Stuck	66	0	712	0	1

Table 7.3: Number of all result pairs for the baseline (disable) and detection (check) with explicit analysis and full pruning.

7.2.2.1 Detection for Explicit Value Analysis

It was already visible that the runtime detection uncovered a substantial amount of cases, where refinement progress was halted and stopped these executions with an inconclusive ("Stuck") result.

What is not visible on the table above is what result the verification would have given to these tasks if not runtime monitoring was used. Table 7.3 answers this question as well. The columns depict result values for the baseline executions and the row depict result values with detection – the numbers in the cells show how many input tasks have taken up this combination of results for these two combinations.

The majority of tasks stopped by the detection would have been timeouts (712), while some would have been technical errors (66). Most of the rest of the results are the same in both cases.

There is only a single task that would have a successful (true) result without the detection. After manually analysing this task, I found that it was a rare case of the SMT solver being non-deterministic inbetween iterations – with other words solving this task was a "lucky error".

7.2.2.2 Mitigation for Explicit Value Analysis

Table 7.4 is similar to Table 7.3, but it compares the baseline to the mitigation instead.

The values in the two tables are somewhat similar, however there are some key differences:

		disable				
		ERR	OOM	T/O	False	True
mitigate	ERR	964	0	43	0	0
	OOM	0	4	0	0	0
	T/O	2	0	2308	0	0
	False	1	0	90	194	0
	True	0	0	0	0	217
	Stuck	0	0	17	0	1

Table 7.4: Number of all result pairs for the baseline (disable) and mitigation (mitigate) with *explicit analysis and full pruning*

- there are only 17 tasks that were stopped be the monitoring, while above there were 712,
- but the number of successful, false results is now 90 instead of 0 – which is a 122% improvement in number of successful verification compared to the baseline,
- unfortunately the number of "remaining" timeouts, i.e. where both timed out, also went up from 1748 to 2308.

7.2.2.3 Differences in Execution Time

After examining the results of the verification executions, there is another variable worth investigating: the wall time of executions. As described in Chapter 6, the advantage of the runtime detection is sparing time by stopping the execution instead of it timing out. However, this is only really advantageous, if the execution is stopped fairly early.

Figure 7.1 shows the density of tasks stopped by the monitoring in 20 second interval bins. Density in this case is calculated by dividing the number of occurrences in the given bin by the product of the width of the bin (20) and the total number of data points.

Both combinations stop the executions mostly in the first 20 seconds, but with mitigation there are some bins with high density much later on as well – in these cases mitigation probably tried to "save" the execution, but failed.

In the end, using detection only was way faster, as the summarized walltime of the baseline execution was 332 hours, while it was 314 hours with mitigation and only 244 hours with detection, which is 73.5% of the baseline execution time.

RQ1 Conclusion Choosing what runtime monitoring methods to use is not an easy task and has to be done on a case by case basis. However, the following pointers can be utilized:

- if time is a really important factor, using runtime detection, but not the runtime mitigation, might be more beneficial, especially if other techniques, such as a sequential portfolio, trying other CEGAR configurations afterwards is also used,
- if time is not such an important factor, using the mitigation technique will be able to provide an improvement in the number of successes, but most likely only a much smaller improvement regarding execution time.

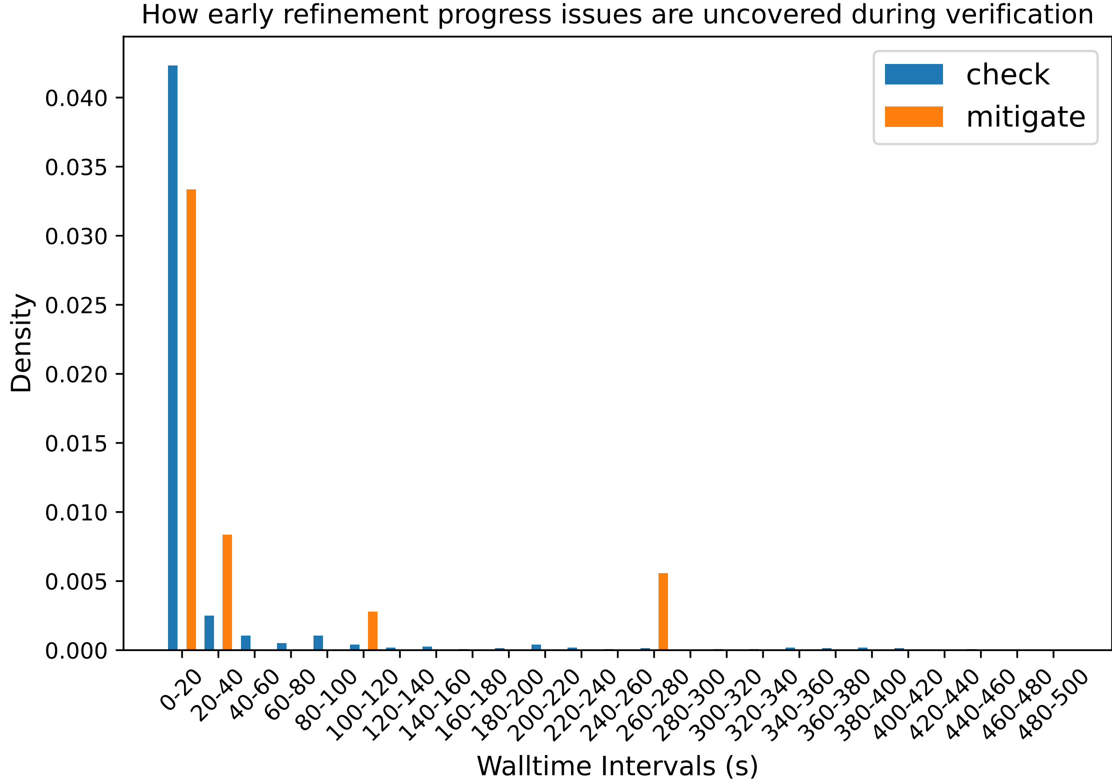


Figure 7.1: Histogram showing how early verifier executions are stopped due to lack of refinement progress, if using detection only (*check*) or mitigation as well (*mitigate*). Density is calculated for 20 second bins and normalized.

7.2.3 RQ2 - Predicate Analysis

The results for Cartesian predicate analysis are summarized in Table 7.5. There are only 9 cases of refinement progress detection stopping the analysis ("Stuck"). One is the already known input program, which induces solver non-determinism with the explicit analysis as well, as described above in Section 7.2.2.1.

There were only 2 input programs that got caught by detection and timed out without it, based on which we can fairly safely assume that refinement progress really stopped.

However the other 6 input programs ended up in different technical errors without detection. This is unfortunate, as this way we can not be sure if they would have timed out or would have been successful without technical issues.

RQ2 Conclusion All in all, compared to the 713 executions of explicit analysis (see Table 7.3) that were caught and stopped, the number of executions caught here is negligible.

These results strongly indicate that the main issue behind refinement progress stopping is low expressive power and thus the issue barely concerns abstract domains with larger expressive power. Of course, it is possible that heuristics with large information loss exist that could cause this issue as well, but in this experiment and in Theta this seems unlikely.

However, as the results of the baseline were not affected negatively by the monitoring techniques, it is unnecessary to turn it off. It might even be beneficial when experimenting with new configurations to see if timeouts are hiding an "infinite CEGAR-loop" or not.

		disable				
		ERR	**OOM**	**T/O**	**False**	**True**
check	**ERR**	1112	0	0	0	0
	OOM	0	4	0	0	0
	T/O	0	0	2069	1	0
	False	0	0	0	232	0
	True	0	0	0	0	414
	Stuck	6	0	2	0	1

Table 7.5: Results of *Cartesian predicate analysis*: number of all result pairs for the baseline configuration (*disable*) and monitoring (counterexamples only) (*check*).

7.2.4 RQ3 - Tracking ARGs

Figure 7.2 shows the number of different outputs for the explicit analysis with lazy pruning and with full pruning as well – this time only detection is experimented on.

Comparing the results of explicit analysis with lazy and full abstraction, the plots look somewhat similar: there is again a large number of executions stopped by detection due to refinement progress issues – however, the number of stopped executions is significantly higher with lazy pruning, which is surprising. For a yet unknown reason lazy pruning seems to be more prone to this issue than full pruning.

The research question however was the difference inbetween results with and without tracking the ARGs and precisions when using lazy pruning. As expected and shown in 7.6a, there were 0 false positive detections, i.e. input tasks that were "Stuck" with *check*, but successful (False or True) with *checkarg*, when using full pruning. However, there were only a 4 false positives with lazy pruning as well, which might be negligible in most cases (shown in 7.6b.

Another cell in the tables needs explanation, which is the Stuck/Timeout (check/checkarg) combination, which has the value 3 and 6 for the tables. With lazy pruning it would be possible that the execution timed out while trying to "mend" itself, but for full pruning this should not be possible. However, manually checking the logs for these 9 executions, the explanation is much simpler: the "Stuck" results were given fairly close to timeout and tracking the ARGs was a bit slower and timed out before giving the same result. The "mirrored" values in the tables are 0, which allows the assumption to a really slight extra overhead when tracking more than just counterexamples.

RQ3 Conclusion False positive detections for lazy pruning are possible, but not common. If false positives are absolutely unacceptable, e.g. when trying to verify a single input task and getting a successful result is the main priority, it might be worth to track everything, but it typically makes little difference, while also adding a slight overhead.

However, the experiment shows that false positives are *possible* and might be possible or even more common with other CEGAR configurations as well, as lazy pruning is just an example of many other heuristics and options.

		checkarg					
		ERR	**OOM**	**T/O**	**False**	**True**	**Stuck**
check	**ERR**	901	0	0	0	0	0
	OOM	0	4	0	0	0	0
	T/O	0	0	1745	0	0	0
	False	0	0	0	194	0	0
	True	0	0	0	0	217	0
	Stuck	0	0	3	0	0	776

(a) *Explicit abstraction with full pruning* with/without tracking the ARGs and precisions

		checkarg					
		ERR	**OOM**	**T/O**	**False**	**True**	**Stuck**
check	**ERR**	896	0	0	0	0	0
	OOM	0	4	0	0	0	0
	T/O	0	0	1265	0	0	0
	False	0	0	0	178	0	0
	True	0	0	0	0	167	0
	Stuck	0	0	6	4	0	1320

(b) *Explicit abstraction with lazy pruning* with/without tracking the ARGs and precisions

Table 7.6: Number of all result pairs with *explicit analysis and lazy/full pruning*, with-/without tracking the ARGs and precisions

7.3 Conclusion

Based on the results, the runtime detection technique can significantly decrease the number of timeouts and thus decrease the overall execution time of verification benchmarks if using explicit value analysis.

However, on abstract domains with higher expressive power, such as predicate analysis, it might not add significant improvements, as refinement progress rarely halts in those cases.

Furthermore, the detection technique proved to not be prone to false positive detections with the analyzed CEGAR configurations, even if only tracking counterexamples.

The mitigation technique might help in solving more tasks, but spares much less time than just using detection. Using a sequential portfolio of more CEGAR configurations might prove more useful instead in some cases.

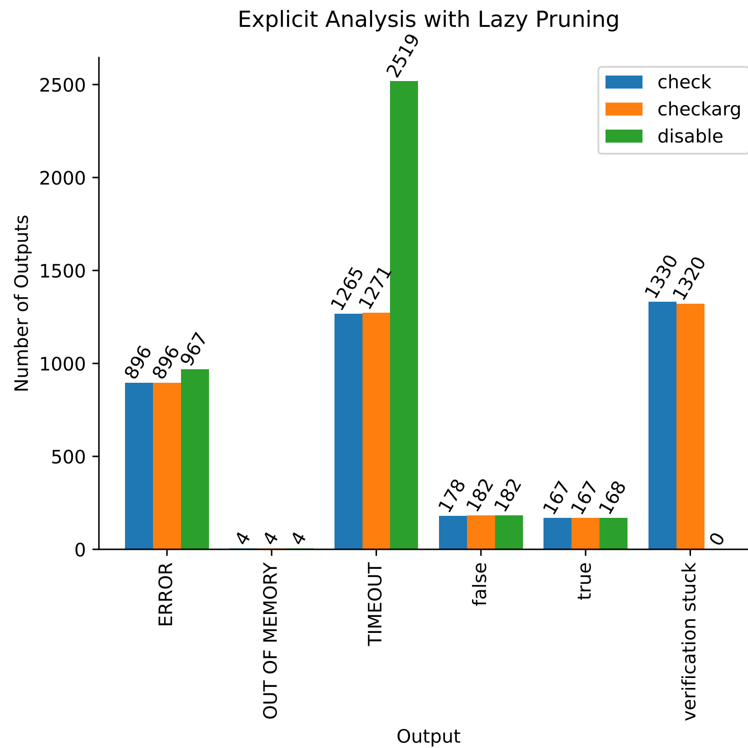
7.3.1 Threats to Validity

Internal Validity The accuracy of measurements is insured by using the BenchExec framework [23]. Further stability is accomplished by executing the same experiments twice and removing input programs that the tool achieved deviating results on. Some deviation and uncertainty might still be present regarding execution time and memory consumption, but these metrics were rarely or not at all used in the evaluation.

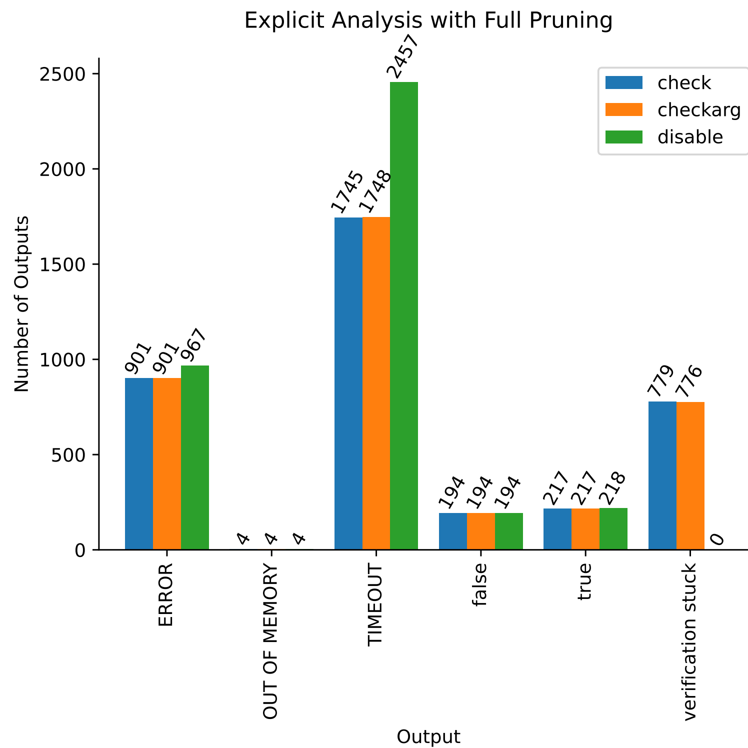
Construct Validity Construct validity ensure that the right metrics are used to measure the examined property. The point of verification is to successfully prove safety or fault in the input program, thus the output of the analysis is the most important metric most

of the time. In this case, other outputs, such as the detection of no refinement progress, might also be deemed a success, depending on what we want to achieve, but this does not change the fact that the main metric is the output of the tool. A secondary metric is execution time, as the time spared by stopping "hopeless" analyses can be immensely helpful in practice.

External Validity External validity is concerned with how well the results can be generalized. This is an important topic regarding this evaluation, as the number of CEGAR configurations used is only a small sample of all the possible ones. However, many of the main conclusions shows the possibility of some events, rather than their certain occurrence, i.e. it is shown that the technique can be quite successful with at least one abstract domain. However, further generalizability would probably only be possible with more broad benchmarks on more configurations.



(a) Results of explicit analysis with lazy pruning, without detection (*disable*), with detection while tracking only counterexamples (*check*) or ARGs and precisions as well (*checkarg*)



(b) Results of explicit analysis with full pruning, without detection (*disable*), with detection while tracking only counterexamples (*check*) or ARGs and precisions as well (*checkarg*)

Figure 7.2: Plots comparing results of explicit analysis with lazy or full pruning

Part III

Related Work and Conclusion

Chapter 8

Related Work

8.1 The Landscape of Verification Tools

Both parts of this work build upon the abstraction capabilities of the tools used, thus it is important to examine how common abstraction is in verification 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 [12][11] 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 [12], 11 of these tools use Counterexample-guided Abstraction Refinement (CEGAR) [32], 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 [54] 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 [54]. One of the competitors is LoLA [67][75], 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 their report slides, some tools also apply abstraction here as well. For example nuXmv [28], 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`

8.2 Test Generation with Model Checkers

Fraser et al. [41] describes several tools and papers about generating test cases with model checkers. Many of the works cited in this survey [39, 43] and even more recent works [56] 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. [40] 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.

8.3 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 [60], UML statecharts to Petri nets [73] or BPMN models to Petri nets [59], verifying properties, such as termination and structural properties.

Varró and Pataricza [73] 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 work can be viewed as a complementary extension of the works mentioned above.

8.4 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 [64] 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 [38].

Test generation and conformance test suites are commonly used in the testing of compilers [29]. Test generation most commonly builds on the grammar of the programming language. However, ambiguous or non-deterministic executions are rarely tested.

8.5 Heuristics and Optimizations in CEGAR

High-Level Portfolios Many state of the art verifiers employ their own portfolios [12], but there is existing work of even higher-level portfolios in the form of the CoVeriTeam framework, which enables its user to create portfolios out of well-known verifiers [13], raising portfolios above the level of the tools themselves. This trend of portfolios becoming more and more important motivates my work, as stopping "hopeless" executions saves CPU time for other tools and configurations.

Earlier Work in Theta One of the main traits of Theta is configurability. This is maybe best shown in the paper "Efficient Strategies for CEGAR-Based Model Checking" [49], which details and evaluates several CEGAR configurations and optimizations, some of which were also used and played an important role in my evaluation in 7.

CPAChecker Besides their own portfolio including several algorithms, CPAChecker also utilizes several interesting heuristics and optimizations: they include another, slightly different version of lazy pruning, use BMC to double check results and so on [15]. The tool itself includes even more heuristics, which I do not think are published anywhere, for example besides time limits, counterexample limits are also often utilized and they also try to get new refinements, if the first one they get is known. These are somewhat similar to my techniques, although they are separate heuristics and the tool does not appear to track counterexamples or stop the analysis with inconclusive results.

Ultimate Automizer [50][51] This software verification tool uses an automata-based CEGAR-scheme. They use runtime algorithm selection techniques inside the refinement algorithm, more precisely they are dynamically changing between several SMT solvers and their configurations to create the "best" possible interpolants based on the ones created in the earlier iterations. From my viewpoint, this is similar to mitigation techniques. It would be worth considering doing something similar as an SMT solver based mitigation technique after detection.

Chapter 9

Conclusion

9.1 Summary of Results

In this thesis I proposed two different extensions for the CEGAR algorithm. They serve different purposes: trace generation is a tool for the validation of the tool to filter issues in the model transformations, while the purpose of refinement progress monitoring is mainly to improve performance. However, both serve to improve the usability and applicability of model checking in practice.

Trace Generation Chapter 3 described the typical *formal verification process* and formulated the main motivation of this work (*"How can we trust formal verification tools?"*). In Chapter 4, I proposed an algorithm for generating execution traces with model checkers. Lastly, in Chapter 5 I implemented a prototype of the algorithms and the validation process and designed two case studies.

Refinement Progress Monitoring Chapter 6 introduced in detail what causes lie behind refinement progress issues in CEGAR. It also updated my earlier work on detection and mitigation of these issues, while giving more detailed analysis of these techniques. Chapter 7 introduced my implementation in Theta, formulated relevant research questions based on Chapter 6 and answered these through designing, executing and analysing experiments on my implementation with verification benchmarks.

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.

9.2 Future Work

9.2.1 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.

9.2.2 Runtime Monitoring

The runtime detection of this work can be extended with further mitigation techniques. Changing some parts of the CEGAR configuration dynamically during the analysis, such as the solver [50] or abstract domains [19], is not unheard of and might prove interesting to combine such methods with the detection technique of this work.

Combining detection with more portfolios is also an interesting topic. The first version of detection and mitigation was already utilized in the portfolio added to Theta in my earlier work [1], but this was based on an empirical approach. Further benchmarks and updating the portfolio, especially including features Theta was updated with since then, e.g. the ability to verify multithreaded programs, might prove even more interesting.

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. Efficient techniques for formal verification of C programs. Master's thesis, Budapest University of Technology and Economics, 2021.

[2] 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`.

[3] 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.

[4] 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`.

[5] 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`.

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

[7] 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`.

[8] 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.

[9] 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.

[10] 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`.

[11] 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.

[12] 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.

[13] Dirk Beyer and Sudeep Kanav. Coveriteam: On-demand composition of cooperative verification systems. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 561–579, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

[14] 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.

[15] Dirk Beyer and Stefan Löwe. Explicit-value analysis based on CEGAR and interpolation. *CoRR*, abs/1212.6542, 2012. URL http://arxiv.org/abs/1212.6542.

[16] 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.

[17] 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.

[18] 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.

[19] Dirk Beyer, Thomas A. Henzinger, and Gregory Theoduloz. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38, 2008. DOI: 10.1109/ASE.2008.13.

[20] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, pages 20–38, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23404-5.

[21] 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.

[22] 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.

[23] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21 (1):1–29, November 2017. DOI: `10.1007/s10009-017-0469-y`. URL `https://doi.org/10.1007/s10009-017-0469-y`.

[24] 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.

[25] 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`.

[26] 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`.

[27] 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`.

[28] 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.

[29] 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`.

[30] 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`.

[31] 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`.

[32] 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`.

[33] 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`.

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

[35] 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`.

[36] 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.

[37] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 265–281, Cham, 2017. Springer International Publishing. ISBN 978-3-319-57288-8.

[38] Márton Elekes, Vince Molnár, and Zoltán Micskei. Assessing the specification of modelling language semantics: a study on UML PSSM. *Software Quality Journal*, March 2023. DOI: `10.1007/s11219-023-09617-5`. URL `https://doi.org/10.1007/s11219-023-09617-5`.

[39] 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.

[40] 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: `10.1016/j.jss.2009.05.016`. SI: QSIC 2007.

[41] 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`.

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

[43] 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.

[44] 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`.

[45] 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`.

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

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

[48] 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.

[49] Ákos Hajdu and Zoltán Micskei. Efficient Strategies for CEGAR-Based Model Checking. *Journal of Automated Reasoning*, 64(6):1051–1091, November 2019. DOI: 10.1007/s10817-019-09535-x.

[50] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software Model Checking for People Who Love Automata. In *Computer Aided Verification*, pages 36–52. Springer Berlin Heidelberg, 2013.

[51] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate Automizer and the Search for Perfect Interpolants. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–451. Springer, 2018.

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

[53] 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.

[54] 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.

[55] 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.

[56] Daniset González Lima, RauÍ 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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`.

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

[63] 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.

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

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

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

[67] 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.

[68] 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`.

[69] 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`.

[70] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. MUX: algorithm selection for software model checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, 2014. DOI: `10.1145/2597073.2597080`.

[71] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.

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

[73] 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.

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

[75] 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.

[76] Zsófia Ádám and Zoltán Micskei. Runtime monitoring of refinement progress in cegar-based model checking: Dataset, Jun 2023.

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).

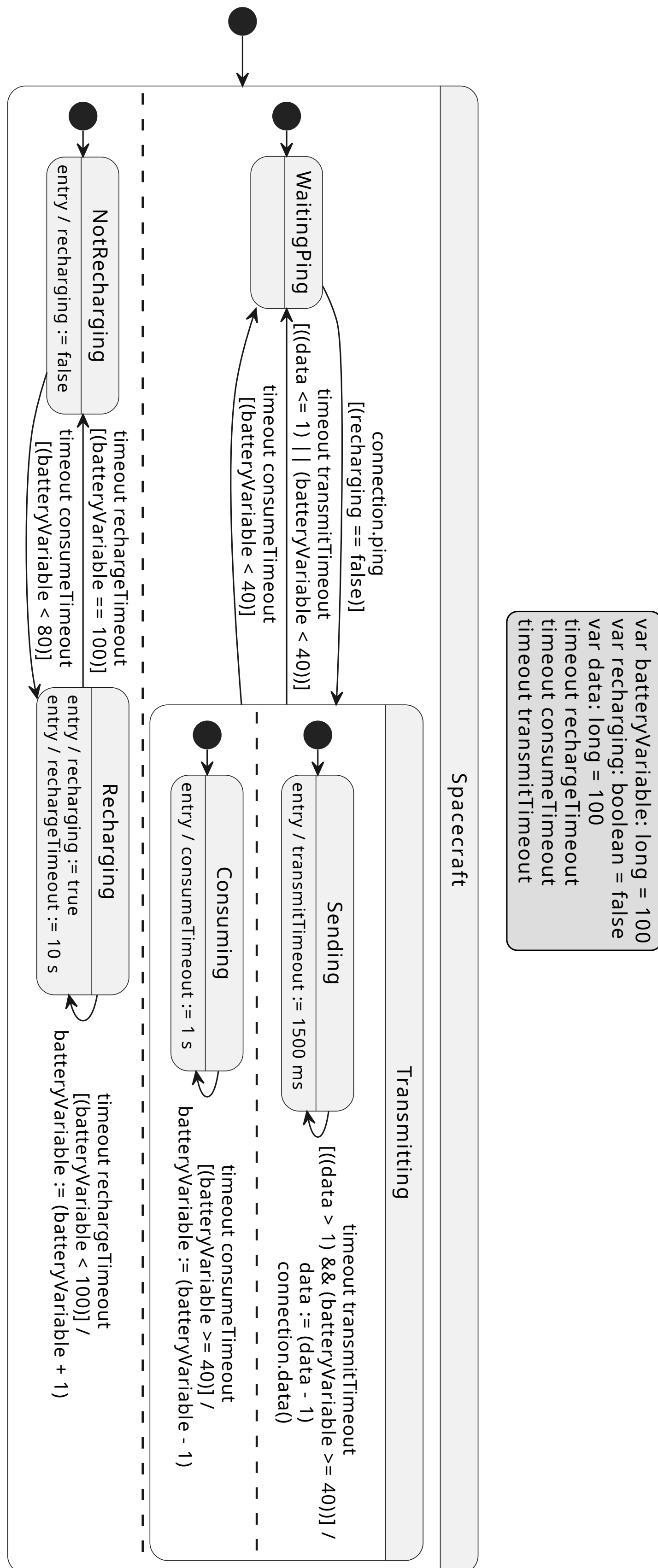


Figure A.0.1: Spacecraft model of the Simple Space Mission case study.

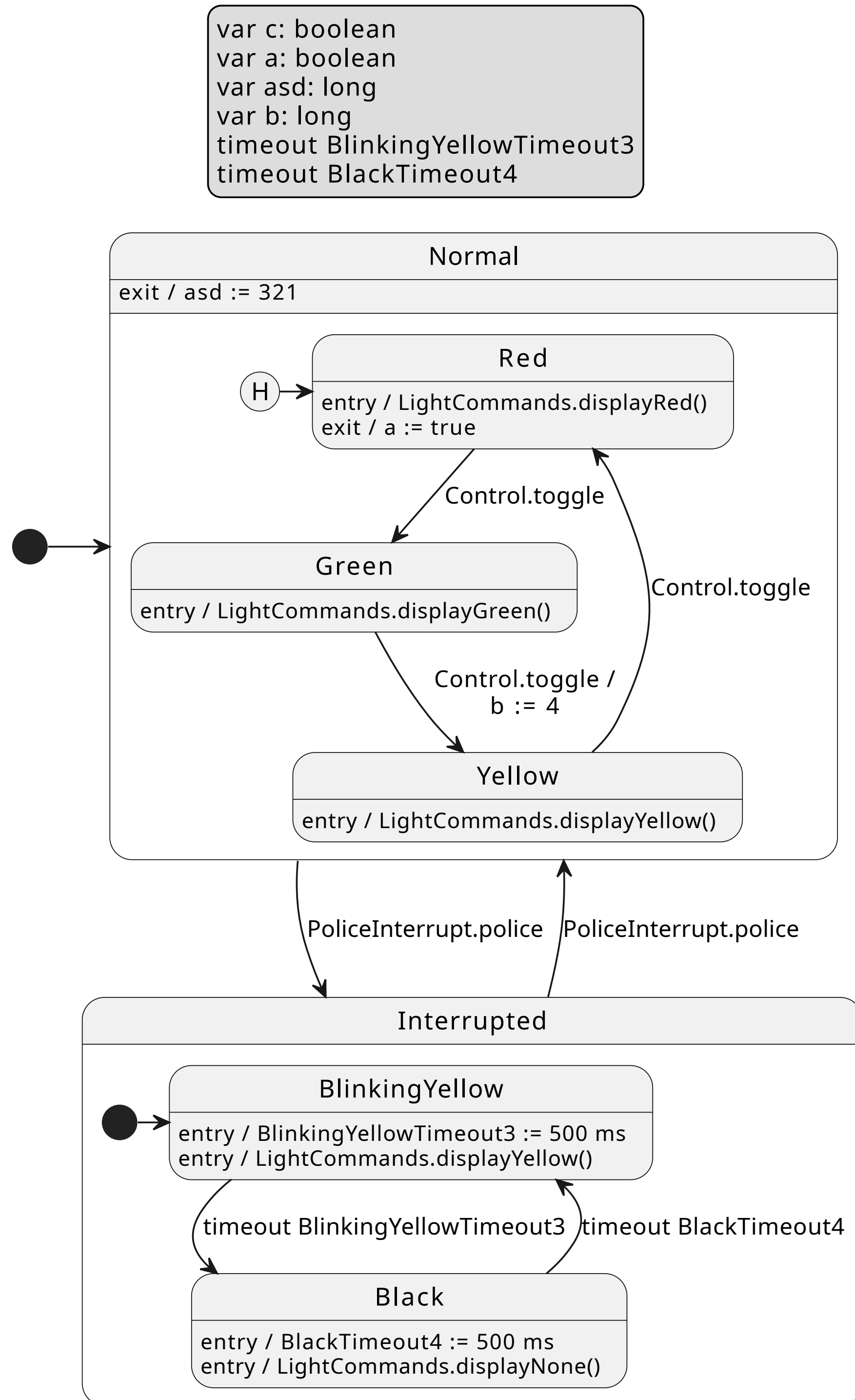


Figure A.0.2: Traffic Light model from the Crossroads tutorial models.

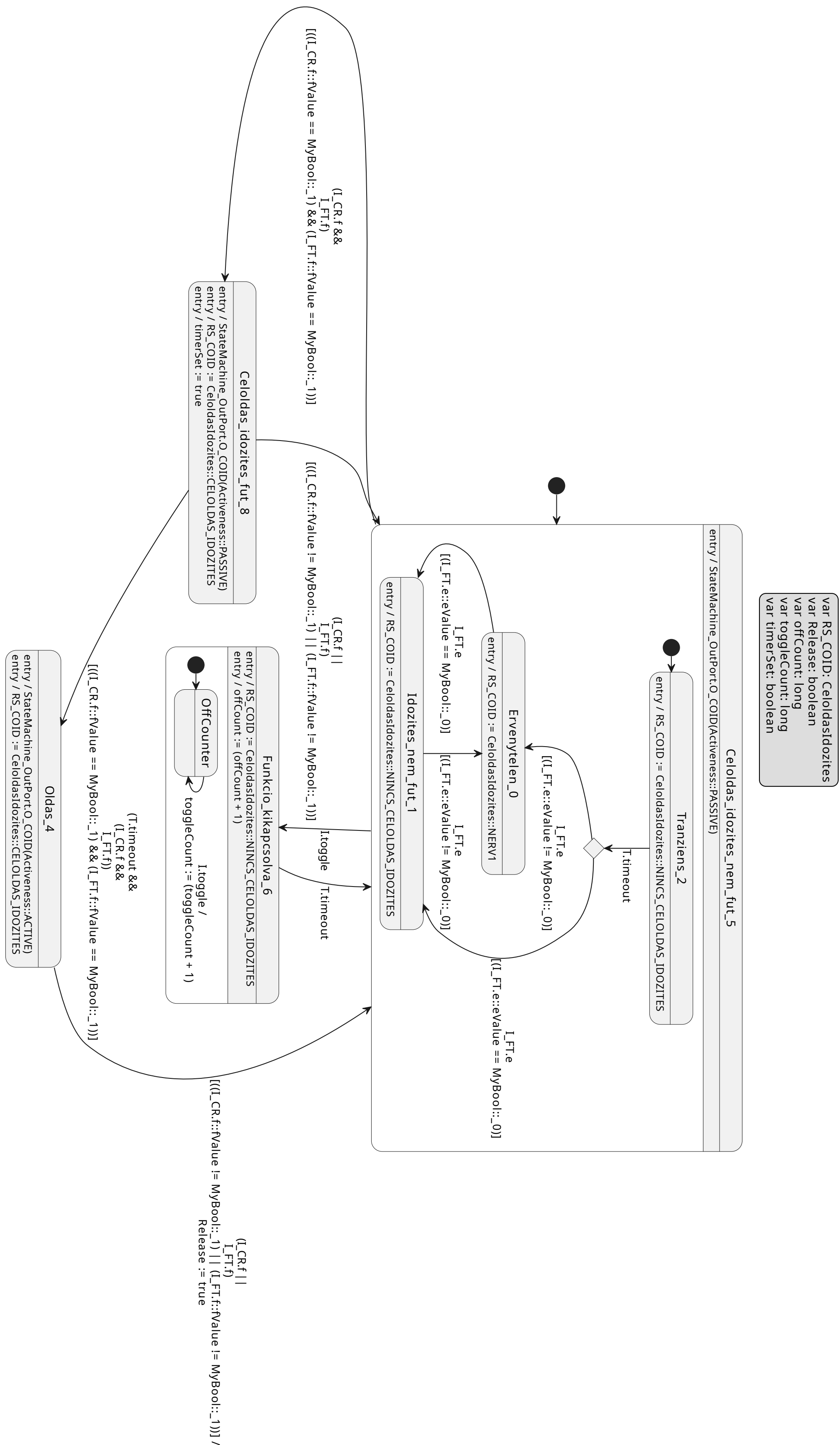


Figure A.0.3: Signaller model of the Railway Traffic Control System case study.