A Thesis Submitted for the Degree of Dr.-Ing.

Self-Verification

Verification of Embedded Systems after Deployment

by
Martin Ring

the 28th of May 2021

Supervisor: Prof. Dr. Christoph Lüth
Second Referee: Prof. Dr. Martin Fränzle

Table of Contents

1 Introduction · · · 1
- 1.1 Self-Verification · · · 2
- 1.2 Structure · · · 4
- 1.3 About this thesis · · · 4
  - 1.3.1 Source code · · · 5
  - 1.3.2 Disambiguation · · · 6

2 A Priori Verification · · · 7
- 2.1 Background · · · 7
- 2.2 Hardware Design Abstractions · · · 9
  - 2.2.1 The Informal Specification Level (ISL) · · · 9
  - 2.2.2 The Formal Specification Level (FSL) · · · 9
  - 2.2.3 The Electronic System Level (ESL) · · · 10
  - 2.2.4 The Register Transfer Level (RTL) and below · · · 11
  - 2.2.5 Different Levels of Abstraction · · · 11
  - 2.2.6 Example: an Access Control System · · · 12
- 2.3 Working with SysML · · · 14
  - 2.3.1 Scope of the Language · · · 15
  - 2.3.2 Syntax · · · 15
  - 2.3.3 Semantics · · · 17
  - 2.3.4 Reference Compiler · · · 17
- 2.4 A Framework for Change Impact Analysis · · · 17
  - 2.4.1 Related Work · · · 17
  - 2.4.2 Underlying Semantics · · · 19
  - 2.4.3 Semantic Relations Across Specification Levels · · · 22
  - 2.4.4 Syntactic Representation · · · 23
  - 2.4.5 Syntactic Difference Analysis · · · 24
  - 2.4.6 Semantic Difference Analysis · · · 24
  - 2.4.7 Change Impact Analysis · · · 27
- 2.5 Reasoning about OCL · · · 28
- 2.6 A User Interface for Change Impact Analysis · · · 29

2.7 Conclusion · · · 32
3 Fundamentals of Self-Verification · · · 33
3.1 General Idea · · · 34
3.2 Implementation · · · 37
3.2.1 The Design Process · · · 38
3.2.2 The Design Process At Work · · · 39
3.3 Evaluation · · · 45
3.3.1 Evaluation · · · 46
3.3.2 Practical Exploitation · · · 48
3.4 Discussion · · · 50
3.5 Conclusion · · · 51
4 Design of Self-Verifying Systems · · · 53
4.1 Self-Verification, Design Time & Run-time Verification · · · 53
4.2 Case Study · · · 56
4.2.1 Informal Description · · · 56
4.2.2 Formal Specification · · · 58
4.2.3 When to Verify · · · 61
4.3 Realization · · · 63
4.3.1 Applying the Design-Flow for Self-Verification · · · 63
4.3.2 The Demonstrator · · · 65
4.4 When to Prove · · · 69
4.5 Conclusion · · · 70
5 Proof Partitioning · · · 71
5.1 Fixing Free Variables · · · 72
5.2 Verification Run Time Analysis · · · 76
5.3 Proposed Solution · · · 78
5.4 Implementation · · · 81
5.4.1 Evolutionary Algorithms · · · 81
5.4.2 EA-based Verification Run Time Analysis · · · 82
5.5 Experiments and Results · · · 84
5.5.1 Set-up · · · 85
5.5.2 Considered Benchmarks · · · 85
5.5.3 Obtained Results · · · 86
5.5.4 Further Discussion · · · 89

5.6 Conclusion · · · 90
6 The Future of Self-Verification · · · 91
6.1 Predictive Self-Verification · · · 91
6.2 Just-in-Time Verification · · · 92
6.2.1 Prerequisites · · · 94
6.2.2 Operation · · · 94
6.3 Dependent Operation · · · 97
6.4 Verification Aware Inference · · · 98
6.5 Conclusion · · · 99
7 Conclusion · · · 101
7.1 Contributions · · · 103
7.2 Future Work · · · 104
7.3 Conclusion · · · 105
References · · · 107

Disclaimer

I hereby declare that

- this dissertation is my own original work,
- it has been completed without claiming any illegitimate assistance and
- I have acknowledged all sources used (both, verbatim and regarding their content).

Martin Ring, the 19th of April 2021

1 Introduction

Contemporary embedded and cyber-physical systems have become so commonplace that we, almost unconsciously, rely on their correct functioning — we just expect our smartphone, our car, our home appliances to work. This is contrary to the fact that these systems have reached a complexity where the verification of their correct behaviour becomes prohibitively expensive. In the past decades, the verification of embedded and cyber-physical systems has become a pressing, complex and elaborate problem for which a number of high-end tools are available [5]–[8]. Designers and verification engineers have access to an enormous amount of computational power, e.g. in terms of high-end design and compute servers. Yet, time-to-market constraints pressurize the early release of products. As a result, full correctness proofs are often reserved for only the most safety-critical systems. For all other devices, errors during the design process may remain undetected in the final product.

The exponential nature of the problem is fundamentally tied to the combinatorial explosion in verification scenarios [7]. Each year, more complex systems are being designed and need to be verified. As systems incorporate more components and interconnections, the number of potential states and behaviors to verify grows disproportionately. Moore's Law, which observes a doubling in transistors in integrated circuits approximately every two years [9], exemplifies this escalating complexity. This vast increase in elements and their interde-

pendencies challenges traditional verification processes [10]. Iterative improvements have been proposed in the past years, e.g. the introduction of higher levels of abstraction for design such as the *Formal Specification Level* [11] and the *Electronic System Level* [12], or the lifting of SAT solvers to solvers for *SAT Modulo Theory* (SMT) [13]–[17], but these cannot and will not be able to cope with the complexity. The consequences are evident today: While several years back, the actual implementation process was the core activity in any design flow, verification dominates today. In fact, more than 40% of the time and costs within the design are devoted to prove the correctness of a system [18].

Because of this situation, we are convinced that verification cannot solely be addressed by incremental improvements of existing approaches anymore, but rather a shift in the existing verification paradigm. In this thesis, we are proposing a methodology towards such a paradigm shift.

1.1 Self-Verification

Current verification techniques such as theorem proving, model checking, static analysis or testing are conducted at design time and finished before deployment, for two reasons: firstly, we want to make sure the system has no errors before putting it into operation, and secondly, it is not entirely clear how to conduct verification at runtime. But this approach has the drawback that the time for verification is limited; errors which are not caught by the time the system is going into operation will remain undetected and may later on have unintended, unpleasant, or even catastrophic consequences.

On the other hand, verification does not necessarily need to terminate with the end of the development. In *run-time verification*, we check whether a particular run of the system satisfies desired properties. This has the advantage that we do not need to stop verification if we deploy the system, and checking whether a specific run of the sys-

tem satisfies the desired property is of lower complexity compared to model-checking [19]. The drawbacks are that it may be costly to continuously monitor the behaviour of the system at run-time, and that discovering an error post-deployment can often mean that corrective measures are either limited or potentially more costly to implement. This is particularly true for hardware, and systems where the split between hardware and software is decided rather late in the development process.

The idea of *self-verification* as envisioned in [20] is to investigate the middle ground in between: verify properties of the system as soon as practically possible, but as late as necessary. In other words, verification does not terminate with deployment, but is also not deferred until the last moment. The authors of [20] name three benefits that self-verification yields:

(1) *More resources* – the computational power and verification effort of thousands of deployed devices may be combined.
(2) *More time* – the deployment of a system does no longer mark an end to the verification.
(3) *More information* – the environment of a deployed unit becomes concrete and by this can substitute abstract variables with definitive observations.

Of these aspects, the scope of this work is the latter: How can information gained during operation be utilised to speed up the verification process so drastically that it becomes feasible? This thesis is *not* concerned with the former two aspects and does not investigate how computing power and time of deployed systems may be combined. Contrarily, we assume less computational power and time, as we aim to prove properties during normal operation on individual devices with far less capabilities than a dedicated compute server has.

Structure 1.2

The thesis is structured as follows:

- Chapter 2 gives a brief overview of the state of the art in specification of cyber-physicial and embedded systems and introduces advanced concepts of a-priori verification which we build upon in the following chapters.
- Chapter 3 introduces and evaluates a simple scheme that can be applied to postpone parts of a proof into run-time.
- Chapter 4 dives deeper into the impacts self-verification has on the development and how design decisions should be made.
- Chapter 5 introduces a methodology to analyse proofs with respect to the question, which parts offer the most reduction in prover run time when instantiated during system run-time.
- Finally, we sketch some advanced ideas for future work in Chapter 6 and
- conclude with a brief summary of the results in Chapter 7.

About this thesis 1.3

This cumulative thesis is based on the following original publications:

[1] M. Ring, J. Stoppe, C. Luth, and R. Drechsler, “Change impact analysis for hardware designs — from natural language to system level,” in *Forum on Specification & Design Languages (FDL 2016)*, Bremen, Germany, Sep. 2016, pp. 1–7

My contribution: I was responsible for the entire implementation process and collaborated closely with the co-authors on the manuscript, leading the process of drafting, revising, and finalizing the content.

[2] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, “Better Late Than Never — Verification of Embedded Systems After Deployment,” in *Design, Automation Test in Europe Conference Exhibition (DATE 2019)*, Florence, Italy, Mar. 2019, pp. 890–895

My contribution: I solely conducted the implementation process and was deeply involved in the writing phase, actively collaborating on drafting, revisions, and shaping the overall narrative. Additionally, I designed and executed the evaluation.

[3] M. Ring and C. Lüth, “Let’s Prove It Later — Verification at Different Points in Time,” in *International Conference on Software Engineering and Formal Methods (SEFM 2019)*, Oslo, Norway, Sep. 2019, pp. 454–468

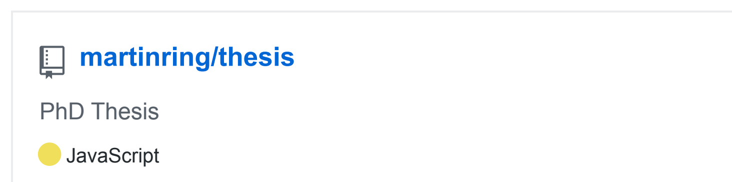
My contribution: I solely conducted the implementation from conception to completion, ensuring the work aligned with our research objectives. My collaboration on the manuscript included leading of discussions as well as drafting, and finalizing of all sections.

[4] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, “Verification Runtime Analysis — Get the Most Out of Partial Verification,” in *Design, Automation Test in Europe Conference Exhibition (DATE 2020)*, Grenoble, France, Mar. 2020, pp. 873–878

My contribution: I was solely responsible for the entire implementation, ensuring its accuracy and relevance to our study. In the writing phase, I collaborated closely with my co-authors, contributing to all sections of the paper. Additionally, I took the lead in designing and executing the evaluation, ensuring our findings were both robust and insightful.

1.3.1 Source code

This thesis is accompanied by a large amount of code, implementing the described concepts, benchmarking these and providing user interfaces for interactive exploration. All code is publicly hosted on the code hosting platform GitHub. Whenever a section is accompanied by an implementation, a link of the following form is provided:



These links can be clicked in the HTML version of this thesis as well as the PDF but obviously not in the printed form. They resolve to a github link of the form `https://github.com/<repo>` where `<repo>` is the name of the repository. E.g. the above link can be accessed as `https://github.com/martinring/thesis`.

> All linked source code was developed solely by the author of this thesis, unless explicitly indicated otherwise.

Disambiguation 1.3.2

We talk about different notions of time in this thesis and are confronted with the ambiguous nature of the term "runtime", "run-time" or "run time". While there exists no clear definition and all three different spellings may be used for every meaning of the word, in this thesis we assign distinct meanings to the terms:

- *run time* is the length of time taken by the execution of a process. E.g. "The run time of the verification was 42 seconds".
- *run-time* is the time at or during which a process runs. E.g. "The property could be proven during run-time". May be used as an attributive adjective as in "run-time information" (referring to information available during run-time).
- finally a *runtime* is shorthand for *Runtime Environment* as in *Java Runtime* or *Haskell Runtime*.

Unfortunately this distinction has not previously been made and hence, the original publication [4] "Verification Runtime Analysis" breaks these rules and should rather be titled "Verification Run Time Analysis".

2 A Priori Verification

To allow thought about the verification of systems after deployment, we will first establish a top-down agile work flow for *a priori* verification in this chapter. To this end, we will introduce the different abstraction levels from informal, naturally phrased specifications down to implementational aspects and show how these can be connected in such a way that requirements can be tracked and verified across different abstraction levels. In the subsequent chapters, this will allow us to choose where self-verification plugs into the flow.

2.1 Background

Traditional hardware design languages (HDLs) such as Verilog or VHDL which are supposed to be synthesised into hardware are increasingly unable to handle large scale designs due to their inherent limitations. For example, they require designers to specify systems to the point where they can be synthesised automatically. The resulting designs need to be built from the bottom up and can only be verified by thorough testing once complete [21], [22]. This approach cannot cope with the shorter design cycles and reduced time to market required in today's marketplace.

The remedy suggested in this chapter is to provide designers with more abstract languages that allow systems to be designed top-down, starting with an abstract model of the system and its requirements.

Several of these languages are being used today. Natural language specifications are the most abstract form of describing a system, allowing the designers to use arbitrary language to explain how the system is supposed to behave and be structured. Formal modelling languages such as the UML or SysML build on a formal definition to avoid the issue of ambiguities in the description. System-level modelling language such as SystemC are the last step before synthesisable HDLs, allowing to build virtual prototypes that can be simulated without actually implementing in the final hardware design.

These languages form a hierarchy and are supposed to be used subsequently: providing a natural language description first, then formalising it, providing a system level model and finally implementing the design at the register transfer level gradually leads designers through the process.

However, when following this approach, several new challenges arise: firstly, we have to keep the models in the different levels of abstraction *consistent* across the different languages and formalisms involved, secondly, we need a uniform notion of *refinement*, and thirdly, we want to be able to *track changes* and their impacts across the different levels of abstraction.

In this chapter we will present a framework which aims at meeting these challenges. The framework provides a uniform management of specifications in these languages at a syntactic level, semantics to relate their meaning (as far as possible) by a notion of refinement, and a comprehensive change management across all levels. We have implemented the framework in a prototype of the *Change Impact Analysis and Control Tool* (ChImpAnC) to demonstrate its principal applicability. It is particularly the change management which makes this approach viable, because we need to be able to handle changing specifications effectively; changes are the norm, rather than the exception, as the design will rarely be correct the first time, and moreover the

tool supported afforded at the more abstract levels will help us to find errors earlier in the design process, necessitating these changes.

2.2 Hardware Design Abstractions

This section gives an overview over different abstraction levels in system design, starting with the most abstract description and successively approaching traditional HDLs.

2.2.1 The Informal Specification Level (ISL)

The most abstract way to describe a system is natural language.

When designing a system, specifying its properties without having to worry about details of mathematical notation and simply using the language one is familiar with instead is a straightforward way to start the design process.

Natural language does not restrict the designer in any way. This openness means that this description cannot be formalised: while natural languages come with grammars that restrict the available constructs, these rules do not mean that the result is an unambiguous description of the system. While natural language processing (NLP) techniques can address some issues, an automatic formalisation of arbitrary text is neither possible nor desired, meaning that these specifications need to be processed manually.

2.2.2 The Formal Specification Level (FSL)

The next step to describe a system in a more exact way are formal languages. Standardised languages such as the Systems Modelling Language (SysML) give designers a way to describe the system readily but at the same time force them to adhere to a formal grammar that makes these descriptions unambiguous [23]. SysML thus offers a way to add precision to the system description.

Still, this formalised notation does not specify all aspects of the system; e.g. the SysML lacks the ability to express non-functional requirements such as timing properties. In other words, FSL models formalise the constraints inherent in the design; e.g. structural diagrams enriched with OCL limit what actions may be performed by the system and how the output values may then be structured. However, while these models may be used to locate potential errors early on in the design process, they are neither complete nor actually executable.

The Electronic System Level (ESL) 2.2.3

The next step in refining the system is to create a working prototype without going into the implementation details required by HDLs. System level modelling languages such as SystemC can describe the behaviour of systems without specifying how this functionality is supposed to be implemented.

SystemC, as the current de-facto ESL standard language [24], allows systems to be described using the C++ programming language while at the same time offering designers the means to describe the structural features of a hardware design. The result is a virtual prototype that can be simulated: parts that are meant to represent hardware are managed by a dedicated simulation kernel which invokes the relevant software parts. This means that the ESL design is much less abstract than at the FSL, representing a model of the system that can be executed, while still being too abstract to be translated into hardware.

However, there are several modern alternatives to SystemC with less commercial traction (for now) such as *Chisel* [25], a DSL embedded in Scala, and *Clash* [26], a language based on Haskell. These share the advantage of having explicit semantic models, which allow for sophisticated static analysis as well as synthesis of lower level RTL models (see below).

In the following we will only consider SystemC (due to its commercial traction and importance in the community) and Clash (for its extensibility and clear semantics).

2.2.4 The Register Transfer Level (RTL) and below

From the ESL, we can map the system design further down to the Register Transfer Level, which gives designers the ability to design systems that may be translated into hardware [27].

Dedicated HDLs are specifically designed to be mapped to hardware, focusing on the description of structural features and parallel execution while at the same time limiting the designer concerning elements that cannot be built as hardware parts such as loops (which need to be unrolled and hence bounded).

Where ESL models can just specify that a module calculates a result using arbitrary means (such as a call to a given software library), RTL designs need to specify how exactly the results are computed.

2.2.5 Different Levels of Abstraction

These different abstraction levels all have particular purposes and use cases:

- Natural language offers a way to quickly come up with an initial description of a given system that is well-readable without prior training and not restricted concerning the described properties;
- FSL models specify system properties in a precise way amenable to formal analysis and reasoning;
- ESL models offer virtual prototypes to be run and tested;
- RTL implementations allow the design to be translated into hardware.

All the different levels describe the same system, yet they are written in different and at first sight unconnected languages. Thus, we need

to ensure that the models at the different abstraction levels are consistent: the natural language requirements need to be represented as formal properties at the FSL, the classes modelled at the FSL need to appear at the ESL in corresponding form etc. Further, one abstraction level may contain several models of the system at different degrees of abstraction: at first, an FSL model should be no more than a translation of the natural language requirements, while a more detailed FSL model should be detailed enough such that we can translate it into SystemC at the ESL. This is called *refinement*: gradually adding more details which constrain the model of the system. Keeping the models throughout the development consistent with each other is called *functional change management*.

Example: an Access Control System 2.2.6

As an example, consider the design of an access control system (appropriated from [28]). It should control the access of people to buildings by controlling the doors. Initial natural language requirements state facts about their relations (Figure 2.1).

```
1  P1: The model is composed of people and buildings
2  P5: Any person in a given building is authorised to be there
```

Figure 2.1 Example - Informal Specification Level

To formalise requirements such as these, we introduce SysML blocks, with added OCL constraints. Here, classes initially include people and buildings; associations include aut and sit, which point to the buildings someone is authorised to enter, or is currently in, respectively (Figure 2.2).

There is one OCL constraint which states that every person can only be in a room she or he is authorised for, i.e. $sit \in aut$. (Figure 2.3).

These formalised but very loose constraints can now be refined further. For example, we introduce doors which connect buildings, and

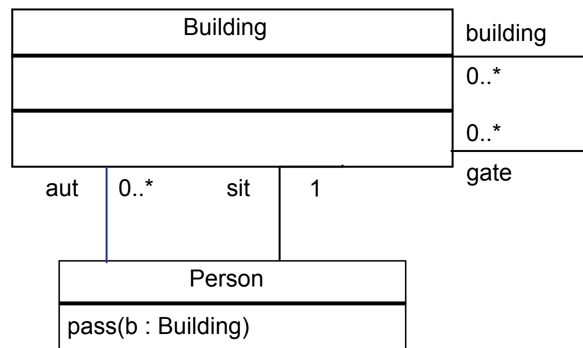


Figure 2.2 Example - Formal Specification Level (block definition diagram)

```
1  context Person
2    inv P5: self.aut→includes(self.sit)
```

Figure 2.3 Example - Formal Specification Level (ocl constraint)

people are authorised to access certain doors. To make this into an ESL specification, we then describe the actual mechanics of operating the door in more detail: when a person is approaching the door, a green or red light should indicate whether access is granted or denied, and a turnstile should open (or not). This can be expressed by a state machine diagram in SysML (Figure 2.4).

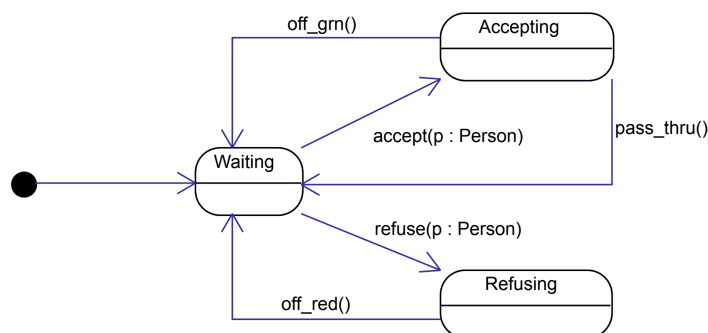


Figure 2.4 Example - Formal Specification Level (state machine diagram)

In our refinement steps, we have replaced modelling classes such as people and buildings by implementation classes like doors. The final refinement step translates a state machine diagram into a SystemC implementation, with doors (but not people) becoming components

(called `SC_MODULE` in SystemC), comprised of a card reader, a turnstile, and green and red LEDs. The turnstile has a method `operate` which implements the state machine diagram above (Figure 2.5).

```
1  SC_MODULE(Door)
2  {
3    //...
4    LED grn;
5    LED red;
6    Turnstile ts;
7    Gate gc;
8  }
9
10 SC_MODULE(Gate)
11 {
12  //...
13  void operate()
14  {
15    //...
16  }
17 };
```

Figure 2.5 Example - Electronic System Level

Working with SysML 2.3

When working with OCL-constrained SysML models, there is a large collection of tools available, which let us design diagrams. These can be classified into two groups. On the one hand those which have an underlying semantic model and on the other hand graphical tools without a semantic model (of SysML). Examples for the former are *Astah SysML*, *Papyrus*, *System Architect* or *Enterprise Architect*. Since we want to formally verify compliance to the specification, we are dependent on the semantic model and thus will not consider the tools from the latter class (e.g. *Microsoft Visio* or *Capella*).

We have chosen to support Papyrus, which is based on the Eclipse Modelling Foundation. However, since SysML is thoroughly specified [29], it should be fairly straight forward to map between different representations.

The reason we chose Papyrus is the fact, that it is the only Framework which allows for semantically meaningful OCL constraints. All other tools we have investigated treat constraints as verbatim text with a language annotation that can indicate an OCL constraint or any other language (another specification language, natural language or a programming language), without the possibility to semantically connect the OCL constraint to the surrounding model.

We have developed our own textual representation of SysML (called *SPECifIC SysML*) which is based on the graphical appearance of diagrams. In this section we give an overview over the language.

2.3.1 Scope of the Language

SPECifIC SysML covers a formally well-defined subset of SysML, in particular it does not support parametric diagrams since they are redundant and can be expressed with OCL constraints. In addition we dropped activity, sequence and use case diagrams, since they only represent test cases and cannot be used to fully specify the behaviour of a system. State machine diagrams are the only behavioural diagrams we support. All other behavioural aspects have to be modelled by means of the OCL. However, since SPECifIC SysML compiles to Papyrus Models it can still be combined with any diagram type that is not supported in the textual representation.

2.3.2 Syntax

The language tries to mimic the appearance of drawn diagrams while keeping it "writable". We use indentation to indicate the structure of a diagram and transfer every textual rule that SysML specifies into the grammar of the language. Most other aspects such as comments, literals and delimiters are taken from the OCL Language specification. Constraints are always assumed to be written in OCL and by this don't require language annotations. In Figure 2.6 the example from Section 2.2.6 is expressed in SPECifIC SysML.

```
1  bdd [package] fsl6::acs [ACS]
2  ---------------------------------------------------------------
3
4  block Building
5    references
6      gate: Building[*] ← building
7        derive: org_dom.dest→asSet()
8      building: Building[*] ← gate
9      org_dom: Door[*] ← org
10
11 block Person
12   operations
13     admitted(q: Door): Boolean { query }
14       post P17: q.org = self.sit and self.aut→includes(q.dest)
15   references
16     aut: Building[*]
17     sit: Building[1] { subsets aut }
18
19 block Door
20   values
21     green: Boolean
22       derive: dap→notEmpty()
23     red: Boolean
24   operations
25     accept()
26       pre: not (green or red)
27     refuse()
28       pre: not (green or red)
29       post: red
30     pass_thru()
31       pre: green
32     off_grn()
33       pre: green
34     off_red()
35       post: not red
36   references
37     org:  Building[1] ← org_dom
38     dest: Building[1]
39   owned behaviors
40     state machine EnterBehavior
41       initial state Waiting
42         accept / → Accepting
43         refuse / → Refusing
44       state Accepting
45         off_grn / → Waiting
46         pass_thru / → Waiting
47       state Refusing
48         off_red / → Waiting
```

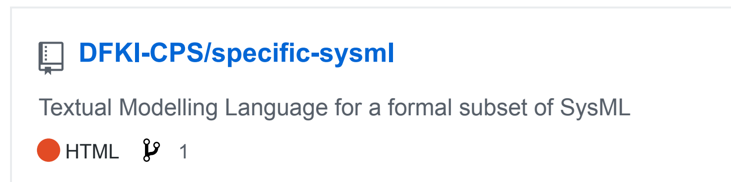
Figure 2.6 The example from Section 2.2.6 expressed in SPECifIC SysML

2.3.3 Semantics

The semantics of the language are completely externalised to the underlying Papyrus framework. Every Diagram expressed in SPECifIC SysML is mapped to a Papyrus SysML Diagram.

2.3.4 Reference Compiler

Further details about the syntax and semantics of the lanugage can be found online in the reference implementation which is freely available. Here, also further tooling around the language can be found.



2.4 A Framework for Change Impact Analysis

Functional change management calculates the impact of syntactical changes using the semantics of the documents. In order to implement it across the different levels of abstraction, we need a unifying semantics for the different levels.

2.4.1 Related Work

Change impact analysis offers more than the currently used source code management (SCM) tools (Git, Subversion, Mercurial, etc.); our work does not compete with any of these but augments them with functional change management, and the proposed solution could be easily integrated into any of these existing SCM solutions.

There are several isolated approaches to functional change management for some of the individual specification levels we described.

EMF itself for example offers a toolset to analyse differences between two models [30], there exists a change management systems for UML diagrams [31], and there is a wealth of techniques on traceability and requirements management [32]. However, these systems share several limitations, the foremost being that there are no semantic connections to external models which could be taken into consideration, leaving the user without knowledge about impacts to other specification layers. Furthermore, we are not aware of any other change management tool available which is able to calculate the impact of changes on the correctness of SysML/OCL refinements. In addition, ChImpAnC supports impact analysis between different abstraction levels.

The analysis of SystemC designs is a complex task that is a research field on its own. Embedding SystemC into a change managed workflow is thus a non-trivial task as a various C++ dialects need to be supported, each tied to compilers that generate an optimised binary version of the design to be run that is stripped of all non-essential meta information. Different approaches to extract the given information include parsing the source code [33]–[37] (which results in the support of only a subset of SystemC, as no existing parser supports all given dialects) or using modified compilation workflows in order to modify the executable design to trace and store the required data itself [38]–[40] (which results in the support of all designs that are built using the compiler being used). In order to keep our approach as applicable as possible, the approach given in [41] was used: instead of relying on the source code, the compiler-generated debug symbols are used. While the format itself differs between compiler architectures, it is always standardised and/or accessible, resulting in a reliable interface to retrieve structural descriptions from SystemC designs.

The OCL approach to specification with preconditions, postconditions and invariants is called design by contract and goes back to [42]. More recently, this approach can be found in component-based design (rich components [43]), or so-called light-weight specification

languages based on a programming language, such as JML [44] for Java or ACSL [45] for C. The latter two focus on what corresponds to the lowest abstraction layer in our setting, the ESL. Existing tools for the whole workflow across all abstraction layers are rare; most closely related are so-called wide-spectrum languages [46] which cover the whole of the design flow. For example, our running example was originally conceived for the B language [47]. Atelier B, the tool supporting B, covers the whole design flow, similar to Event-B, an extension of B with events, which is supported by the Rodin tool chain [48]. Another prominent example is SCADE [49], which supports seamless and rigorous development from abstract specification down to executable software or RTL code by code generation techniques.

The drawback of all these languages and tools is that they tie the user into one language and methodology for the whole design flow, whereas our approach offers designers a best-of-breed approach, and integrates into existing design flows. Moreover, we are not aware of any attempts to apply impact analysis on any of these wide-spectrum languages.

2.4.2 Underlying Semantics

We base our reasoning about the semantics on the reduction to Kripke structures. At its core a Kripke structure consists of a set of states, a transition relation between states, and an associated set of atomic propositions for each state, which hold within that state. By adopting Kripke structures, we are able to encapsulate crucial concepts like state transitions and state-dependent behaviors, offering a foundational basis for verifying properties and understanding the behavior of our system.

Each considered specification level carries its own semantics, shedding light on specific aspects of the system:

Informal Specification Level

The ISL cannot have a mathematically precise semantics, as such would counteract our motivation to use natural language in the first place (we want users to be able to express initial specifications without having to worry about mathematical rigour at the same time). Instead, we use NLP techniques to decompose the natural language requirements into single semantically meaningful requirements, which form the semantic entities at the ISL. Additionally, if NLP does not offer satisfying results, connections between elements of the FSL and the ISL can be drawn manually in order to properly detect the impact of changes across the different abstraction levels.

Formal Specification Level

Our interpretation of the Formal Specification Level (FSL) is based on a semantically well-defined subset of SysML, which includes class, object and state diagrams as well as OCL constraints.

In this context, class and object diagrams provide a notion of state (see [50] for details). Classes describe the system state through an object model, while object diagrams represent specific instances of these states (in particular, the initial states). The transitions between these states are governed by OCL constraints, which specify the conditions under which state transitions may occur.

A formal definition of our FSL subset as introduced in [51], is given by the tuple:

$$SP = \langle \mathcal{M}, init, \textsc{Opn}, inv, pre, post, st \rangle$$

- $\mathcal{M}$ denotes the set of classes within the specification, essentially forming the object model for OCL expressions.
- $init$ specifies the initial states of the system.

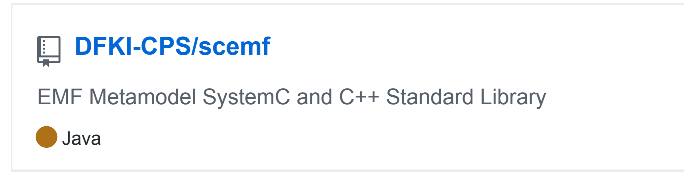
- OPN represents the set of class operations available in the specification.
- inv includes class invariants that must always hold true.
- pre and $post$ are functions that define the pre- and postconditions for the operations in OPN.
- st comprises the set of state diagrams, which are simplified in our formal subset to exclude hierarchical states and concurrent regions. This allows representing state diagrams as pre- and post-conditions over virtual class attributes that track the state.

The semantics of such an FSL specification can be modeled as a Kripke structure, denoted as $[\![SP]\!] = \langle S, I, \rightarrow \rangle$, where:

- S is the set of all possible states of the system.
- I is the set of initial states that satisfy both the conditions in $init$ and the invariants in inv.
- The transition relation $\rightarrow$ encapsulates all permissible state transitions for any operation $o \in$ OPN from one state $\sigma_1 \in S$ to another $\sigma_2 \in S$, under the following conditions:
  (1) All invariants hold in both σ_1 and σ_2.
  (2) The preconditions of o are satisfied in σ_1.
  (3) The postconditions of o are satisfied in σ_2.

Electronic System Level

At the ESL, the semantics are given by the SystemC semantics. States are given by the instances of the SystemC modelling classes (`SC_MODULE` etc.), and transitions are given by the simulation (see [52] for details; however, we use a reasonable abstraction from the concrete SystemC implementation instead of a mathematically precise model of the implementation). Thus, the semantic entities at the ESL are classes, attributes, and methods. For this, we have implemented a semantic meta model for SystemC based on EMF:



Semantic Relations Across Specification Levels 2.4.3

The semantic entities on the respective abstraction levels give rise to notions of mapping between them. From the ISL to FSL and ESL, we map each requirement to one or more specification elements which implement them. Within the FSL, we can utilize a more formal notion of refinement based on the underlying Kripke structures as introduced in [51]:

a concrete specification $\mathcal{C}$ is a refinement of an abstract specification $\mathcal{A}$ if each state transition in $\mathcal{C}$ can be mapped back to a state transition in $\mathcal{A}$, i.e. $\mathcal{C}$ restricts the possible state transitions of $\mathcal{A}$. This refinement can be realised by refining the state (data refinement) or the operations (operational refinement). An example of data refinement is the introduction of new classes or attributes; an example of operational refinement is the implementation of a single operation by a state diagram.

From the FSL to the ESL, we have the usual implementation of SysML diagrams, except that we may map classes in the FSL to instances of the `sc_module` class (corresponding to the fact that in hardware, objects exist more or less *a priori*). Within the ESL (i.e. between two SystemC models) we do not consider refinement, as this would require a more sophisticated semantic modelling of SystemC to consider e.g. timing requirements.

A system development consists of several *layers* $L_1, \ldots, L_n$, which group specifications from one abstraction level. The first layer typically contains the natural language specifications, and the last layer L_n ESL or RTL specifications. Between layers, specifications are re-

lated via refinement: a specification SP from layer L_i is mapped to a specification SP' of layer L_{i+1} if SP' is a semantic refinement. This mapping allows us to keep track of properties; for example, if all initial ISL requirements are mapped to formal properties which are later proven we can be confident that the implementation satisfies the original specifications.

The mappings are mostly constructed automatically (see Section 2.4.7 below), but some have to be constructed by the user (in particular, the mapping of ISL requirements).

2.4.4 Syntactic Representation

The specifications on the different levels are written in different formalisms, each in their own syntax. Since we aim to support a wide variety of file types in an extensible way, it would be inflexible to implement a parser for every concrete input syntax. Hence we decided to employ the widely adopted, generic Eclipse Modelling Framework (EMF) [53], which serves as a common basis for other file types. This means that any format is supported as soon as there is a translation into EMF.

At the ISL, specifications are represented as a list of SysML requirements. At the FSL, we use the SysML tools provided by the Papyrus Framework [54], as well as the EMF OCL representation. For the ESL, we make use of the fact that SystemC models are valid C++ source files, and employ the debug output of the clang compiler to generate an EMF model. The files contain DWARF debug information that can be extracted using the libdwarf/dwarfdump tools. The resulting data is translated to the EMF format using a custom parser/translator. The final result includes namespace and class structures with type hierarchies, operations and attributes.

Syntactic Difference Analysis 2.4.5

The architecture of the functional change management has been derived from previous work in the generic GMoC system [55]. A generic diff algorithm for hierarchical annotated data serves as a basis [56], and provides support for syntactic difference analysis. We adapted this algorithm to operate on generic EMF objects (EObjects). This way we can obtain a minimal set of changes between two EMF files. The GMoC diff algorithm allows us to specify equivalence between the objects; in our case, which attributes identify an object, which orderings have a meaning and which do not. The example in Figure 2.7 states that a SysML block is identified by its name, and that the order of the contained attributes and operations is irrelevant, while on the other hand the order of the parameters of an operation has a semantic meaning.

```
1  element Block {
2    annotations {
3      name!
4    }
5    constituents {
6      unordered { _ }
7    }
8  }
9
10 element Operation {
11   annotations {
12     name!
13   }
14   constituents {
15     ordered { _ }
16   }
17 }
```

Figure 2.7 Example *ecore.equivspec* file

Semantic Difference Analysis 2.4.6

The distinctive feature of the diff algorithm is that it takes the intended semantics of the documents into account. This is achieved by representing the semantics as a graph (*explicit semantics*). The semantic

graph is extracted from the syntactic graph by graph rewrite rules, which can be efficiently implemented in Neo4j; after extraction, the nodes of this semantic graph are connected to the origin nodes of the syntactic tree (Figure 2.8).

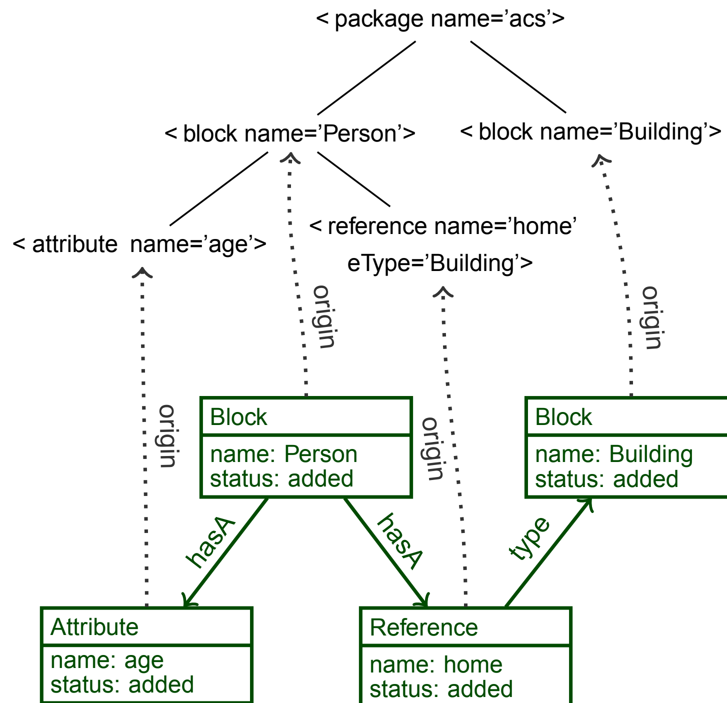


Figure 2.8 Change management via explicit semantics after initial extraction

When a change in an input file occurs, a diff is applied to the syntactic tree. Then, we mark the nodes of the semantic graph as "deleted" (Figure 2.9) and extract the graph again (Figure 2.10). Nodes that are already present in the graph are marked as "preserved", nodes that do not exist are marked as "added", and all other nodes remain marked as "deleted". During this process additional semantic knowledge can be used to handle individual nodes as required.

Thus, we have the *syntactic graph* which consists of the abstract syntax trees, and the *semantic graph* extracted from them. We store both graphs uniformly in the Neo4j graph database, because it allows us to efficiently traverse and transform them while providing superb

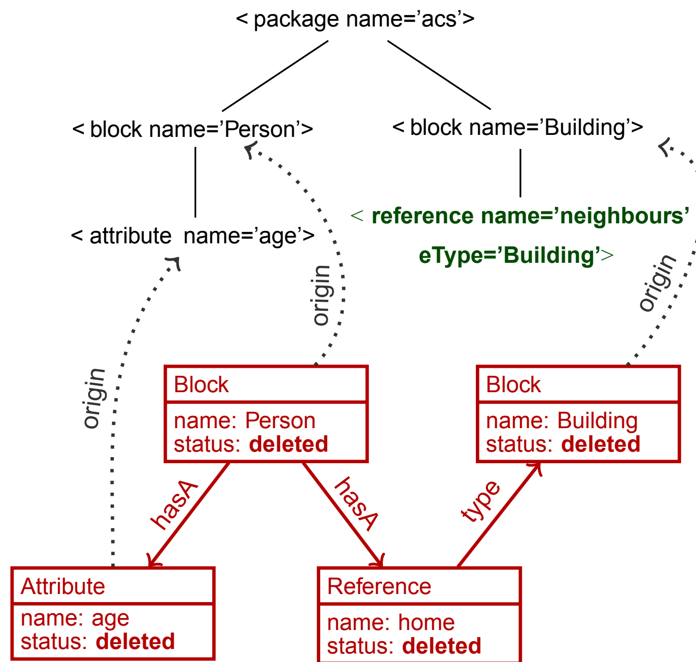


Figure 2.9 Change management via explicit semantics after application of syntactic diff

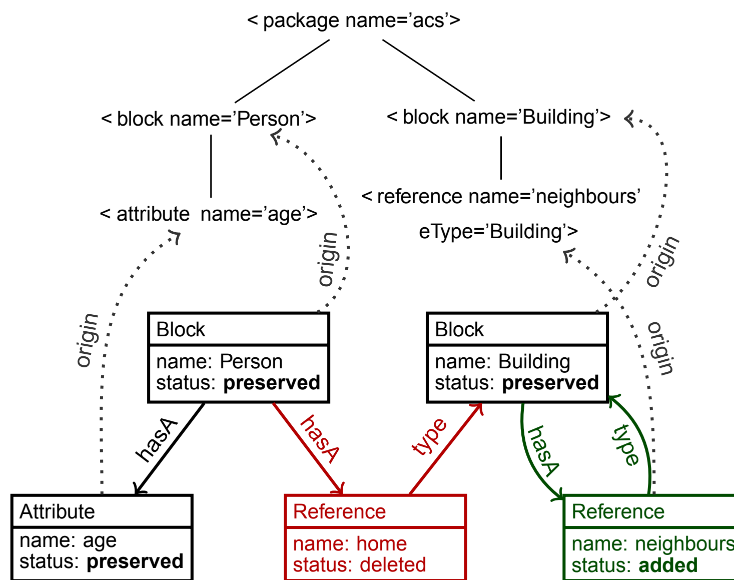


Figure 2.10 Change management via explicit semantics after second extraction

scalability. On top of this we implemented an interface from EMF to Neo4j which allows us to analyse differences between files on disk and the persisted syntactic tree in the database:

2.4.7 Change Impact Analysis

The semantic graphs of specifications from adjacent layers can be mapped semi-automatically by inspecting naming, types and structure of models. Users are always in control of these mappings and can alter or complement them where required to reflect their intentions.

Change propagation follows syntactic changes across the origins along the mappings of the semantic graph. That is, if a syntactic change occurs we find which parts of the semantic graph have their origins in that part of the syntactic graph which has changed, and then check which mappings either point into, or originate from, this part of the semantic graph. To illustrate, consider our example (Section 2.2.6): ISL requirement P5 (left) gets mapped to OCL invariant P5 (middle-left); if either the requirement or the OCL invariant is changed, the other is impacted by the change, as inconsistencies between the two might arise. If the user changes the state diagram in the ESL, this change might impact the ESL implementation, or on the other hand the OCL invariants of the class diagram.

For data or operation refinements, we can calculate the impact of changes more accurately. If we add additional operations to the class `Building` in Section 2.2.6, all data refinements of `Building` will remain valid. The situation gets more complex when we consider the proof obligations that arise from refined OCL constraints. These proof obligations are of the form $c_1 \wedge \ldots \wedge c_n \implies d$, where c_1 to c_n are constraints on the refined level and d is a constraint in the abstract level. If this is proven, we can discharge the obligation and insert additional dependency edges between the constraints $c_1, \ldots, c_n$ and d. If one of these constraints changes the proof will be invalidated and the proof obligation pops up again.

Impact rules such as these are described directly as Neo4j queries; this makes them fast to execute and keeps the impact system extensible.

Reasoning about OCL 2.5

To discharge proof obligations that arise from the formal specification level, we need a method to transfer constraints into lower level representations. For OCL there exists a thorough specification of the semantics [57], however SystemC (and especially its extensions, e.g. TLM) has no such specification, and even the host language C++ is not unambiguous across compilers and platforms.

So Clash (See Section 2.2.3) is a natural choice if we not only want to map and trace changes across layers but also conduct verification (and trace verification results) across layers. For this we have developed a dedicated backend that can translate Clash designs into SMT Bitvector logic (See also Section 3.2.2 Paragraph D). In addition we have built a small tool that can generate the proof obligations as SMT assertions from the SysML Model, the OCL Constraints and the Mappings to the ESL design.

2.6 A User Interface for Change Impact Analysis

In this section, we explore a user interface tailored for change management analysis: ChImpAnC. Developed as part of our approach to a priori verification, this interface serves as a practical tool to illustrate how changes in the system can be analyzed more effectively, offering insights into potential applications and future enhancements.

ChImpAnC is realised as a web interface and can either run locally or on a team server. When users open the application in a browser they get presented a multi-column layout representing the different specification layers (Figure 2.11). The leftmost column is the most abstract one — typically natural language — while every additional column to the right represents a refinement step. There are usually more refinement steps involved than would fit into the user interface, so there is a navigation bar on the top where one can select the layer in focus.

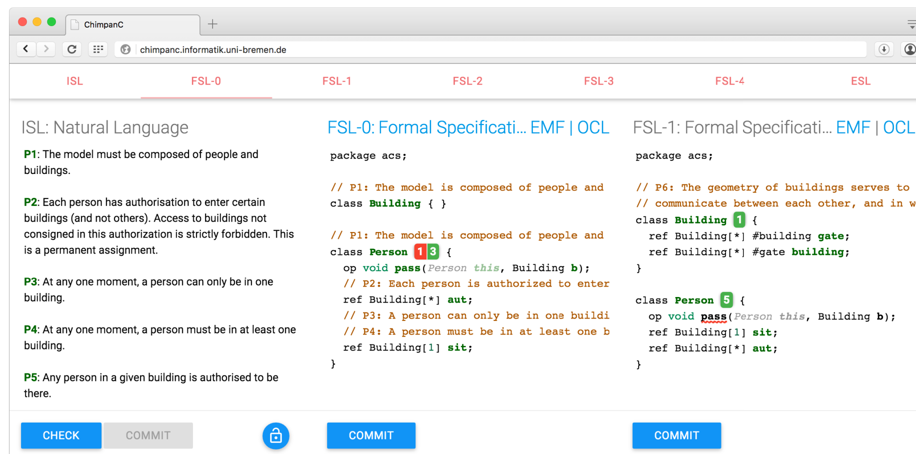


Figure 2.11 The ChImpAnC user interface

Natural language descriptions are treated specially due to the fact that they might be mapped to arbitrary lower specification layers. I.e. it might be intended that an abstract formal description does not contain every requirement described by a stake holder and that the re-

quirements are taken care of in subsequent refinements. They may be locked by clicking on the lock icon on the lower right, such that the user is able to relate the natural language description to lower level refinements.

All extracted model elements are represented as bold identifiers. Mapped model elements appear green. When a user hovers the mouse over such a mapped element, the corresponding refinement is visually emphasised (Figure 2.12).

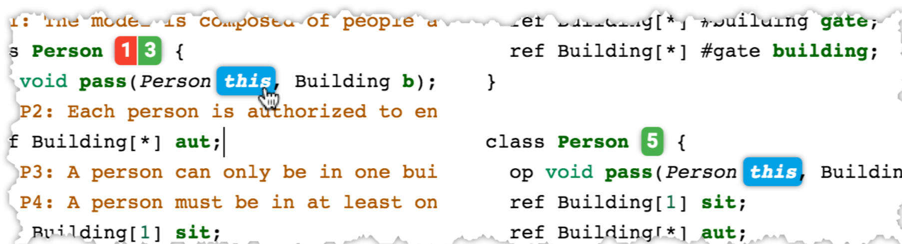


Figure 2.12 Highlighting of mappings

Inconsistencies are highlighted with red wavy underlines. These include elements (abstract models, attributes, references, operations or parameters) which are unmapped in a refinement (Figure 2.13), as well as mismatching mapped types and inconsistent multiplicities of references. In addition, unproven OCL refinements are displayed as a red number next to the respective class definition which indicates the number of open proof obligations. Conversely, discharged proof obligations appear as a green number (Figure 2.14). When the user moves the mouse over a marked element, a tooltip will appear containing information about the inconsistency.

Content warnings are highlighted with orange wavy underlines. These are currently only present in natural language where we automatically rate the quality of refinements, using the techniques from [58]. Again, a detailed description of the warning can be obtained by hovering the mouse over the marked element (Figure 2.15).

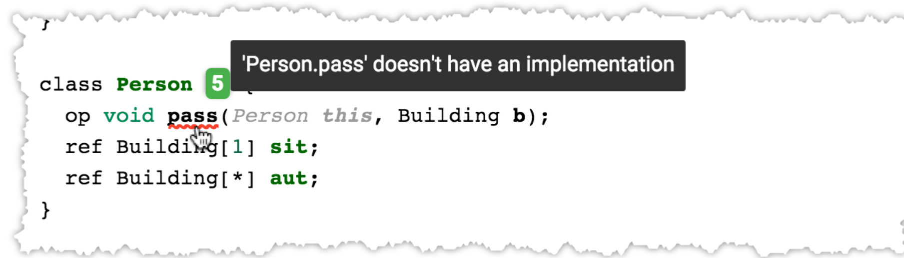


Figure 2.13 Highlighting of inconsistencies

Figure 2.14 Inline display of proof obligations

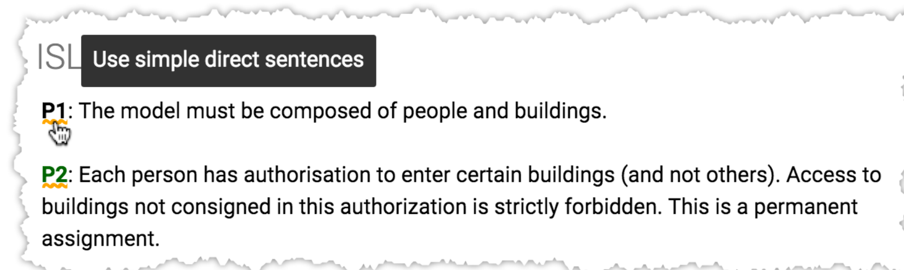


Figure 2.15 A content warning in natural language

Finally, change management support is implemented by impacts. An impact can either indicate that a refinement has changed or that the abstraction has been changed or removed; impacts warnings are the default fallback when there is no automatic solution to propagate a change across layers. It still offers a high value to developers because the possibly affected portions of refinements and abstractions can be narrowed down to small fractions of the specification and inconsistencies can easily be identified. Removed refinements do not trigger an impact warning because they already result in an inconsistent model, and thus an inconsistency error. Impact warnings ap-

pear as orange elements indicating that user attention is required (Figure 2.16).

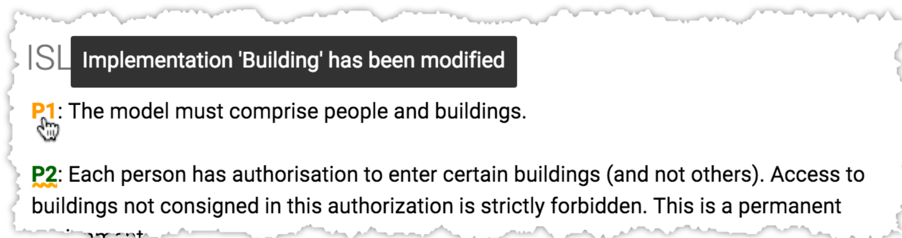
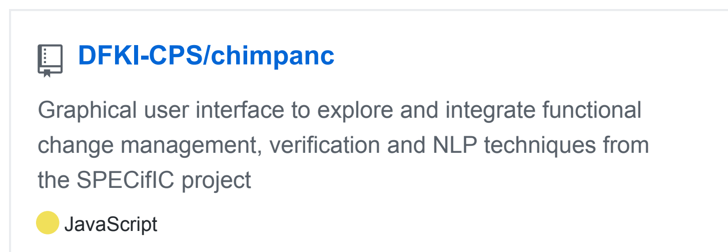


Figure 2.16 An impact warning

2.7 Conclusion

This chapter introduced the different layers of abstraction that may be used to design systems top-down. We presented SPECifIC SysML a textual modelling language that supports a formal subset of the SyML. We integrated all this in ChImpAnC, a tool which supports a comprehensive system design flow across different levels of abstraction, from natural language down to system-level models. ChImpAnC manages the models of the systems at the different abstraction levels, keeps track of dependencies, and calculates the impact of changes. Moreover, it can warn about inter layer inconsistencies that would previously be left unnoticed by the established tool chain. The tool is freely available online:



We will use this foundation to develop the idea of self-verification in the subsequent chapters.

3 Fundamentals of Self-Verification

In this chapter, we propose a simple design and verification methodology which conducts verification after deployment. To this end, we start with the observation that contemporary systems are designed to operate in a variety of operating contexts. In order to do so, *configurations* are used, i.e. parameters which are set post-deployment by the particular environment of the individual system. While these parameters may not change frequently, they are not fixed and hence verification, which, thus far, is conducted prior to deployment, has to consider all possible configurations.

We indicate these configurations as *quasi-static*. Consequently, designers and verification engineers are faced with verifying systems with huge possible search spaces, while after deployment just a fraction is used.

Engineers might restrict the set of allowed configurations at design time in order to reduce the search space and make verification succeed, but this increases costly design time, and runs the risk of excluding possible configurations, decreasing availability, making the system less versatile and hence less marketable than strictly necessary.

Motivated by this observation, this chapter proposes a design and verification methodology which conducts verification after deployment, i.e. in the field and once the actual configuration is observable. Even though it results in continuous verification tasks as the environment keeps changing, the drastic reduction of the search space outweigh this. As a result, embedded and cyber-physical systems can be verified even on a much weaker machine and with much less sophisticated tools, while prior to deployment verification failed due to the exponential complexity.

In order to assess the feasibility of the proposed methodology, we have implemented the proposed design and verification flow and used a lightweight version of the SAT solver MiniSat [59] to solve the resulting verification conditions after deployment. The evaluation of a number of case studies showed that, following the proposed methodology, verification problems which failed prior to deployment (using high-end verification tools and machines) could be completed after deployment using the lightweight solver on reduced hardware.

General Idea 3.1

The key idea of the proposed approach presented here is to defer part of the verification until *after* deployment. At first sight, this seems like a rather strange idea. A system deployed in the field is likely to have far less computational power, memory and network resources available than a design server. However, it has a main advantage which, we argue, outweigh these deficiencies: after deployment, there is generally *more information* about the operating context available.

In order to enjoy this benefit, the design needs to be geared towards verification after deployment. At an abstract level, the general idea is to partition the system state space into one part which changes frequently post-deployment and thus has to be explored symbolically, and one part (preferably as large as possible) which only changes infrequently. This part is called the *configuration*. Marking a variable as

a configuration variable means that its value rarely changes, and entails that we can substitute actual values before verification post-deployment. By marking variables of n bits as configuration variables, we reduce the search space we need to explore for verification by 2^n – turning the exponential growth into an exponential reduction.

The idea and its benefits are illustrated by the following (running) example, which has been deliberately kept simple in order to keep the focus on the methodology.

Example 3.1

The simple light controller system sketched in Figure 3.1 works as the running example in the following. This system connects a controller to a luminosity sensor and a light switch. The controller should turn on the light if the sensor e drops below a given level e_{lo}, and turn it off if it exceeds a given level e_{hi}. To avoid a flickering effect when the luminosity varies close to a given threshold, the lower and upper threshold levels are not equal (hysteresis), and the system should switch off the light only with a certain delay d.

The threshold levels e_{lo}, e_{hi} and the delay d are configuration variables, and can be changed post-deployment.

Systems like these are designed in a flexible fashion, so that they can be applied in various contexts. For the light controller, the threshold levels and delay are not fixed at design or production time but will be set post-deployment. Hence, in order to verify the correctness of the system, we need to take into account *all* possible configurations, which increases the search space exponentially. It also means that a lot of possible configurations are checked during verification which may never be applied during the system's lifetime. Hence, if we instantiate the configuration variables after deployment and keep only

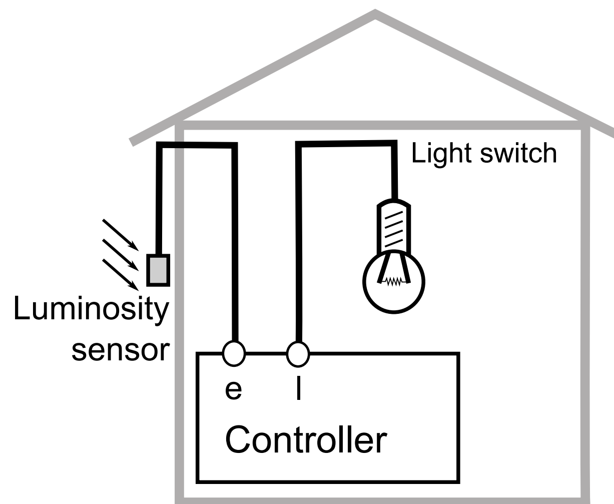


Figure 3.1 Bringing light into darkness: The light controller is connected to a luminosity sensor and switches a light on or off when it becomes too dark or bright.

the variables of the system which change frequently arbitrary, we get a much smaller search space to explore.

The reduced search space can be handled comfortably by a light-weight solver after deployment, even under the prevailing conditions of limited computational resources. But note that this verification is only valid for the particular configuration (i.e. the supplied values for e_{lo}, e_{hi} and d) and, thus, can principally not be done prior to deployment without severely reducing the flexibility and versatility of the system.

Example 3.1 (continued)

Consider again the running example. If we assume a width of 8 bit for the input values (the luminosity sensor and subsequently for the upper and lower bounds) and the time delay, and one bit for the light switch status (these are lower bounds for a realistic system), we get the following search space (where *cnt* is a variable counting up to delay):

$$\begin{array}{cccccccc} e_{lo} & e_{hi} & d & e & cnt & status & & total \\ \underbrace{8 \quad 8 \quad 8}_{configuration} & & & 8 & 8 & 1 & = & 41 \end{array} \tag{3.1}$$

Thus, we need to check an overall search space of 2^{41} states to verify the system, a huge search space for a very simple example.

In contrast, once the system is deployed and applied in the field, the values for e_{lo}, e_{hi} and d rarely change (once when the system is deployed, and afterwards only if the user actively changes the configuration), as opposed to the values of e, cnt and $status$ which vary constantly. Thus, we can mark e_{lo}, e_{hi} and d as configuration variables, and verify the system only when the configuration is changed. By keeping the values of e_{lo}, e_{hi} and d fixed for the verification, the search space reduces to 2^{17} states.

3.2 Implementation

The previous section illustrated the potential of conducting verification after deployment. Based on that, we now describe in detail a possible implementation of this methodology. We first describe the design process in more detail, and then demonstrate it at work with a formal development of the running example considered in the previous section.

The Design Process 3.2.1

The design flow starts with a *modelling phase*, where the structure and behaviour of the system is modelled at an abstract level without referring to any implementation (see Figure 3.2). In our case, we use SysML [29] and OCL [50] to specify the structure and formalise constraints on its behaviour as well as the functional hardware description language Clash [26] for a uniform, executable and synthesiseable model of the system.

The actual specification and implementation languages are of no particular relevance and could be replaced by others (e.g. we could use UML instead of SysML, or SystemC instead of Clash), but serve here to point out the level of abstraction in the corresponding part of the design process.

From the model, we can synthesise an *implementation* of the system by generating a representation in a low-level hardware modelling language such as VHDL or Verilog, which is used to program an FPGA – constituting the actual implementation. Moreover, we want to *verify* that the generated system behaves as specified. In order to do so, we generate a list of *verification conditions* from the executable system model and the specification which have to be shown in order to guarantee this.

Specifically, we translate both the Clash model and the constraints from the OCL specification into *bit-vector logic* (i.e. first-order logic with bit-vectors). Trying to show these in an SMT prover such as Yices [16] or Z3 [17] fails for non-trivial examples, as does trying to show the properties translated into *conjunctive normal form* (CNF) with a SAT solver such as MiniSat. This is where verification usually fails.

However, post-deployment after we have instantiated the configuration variables, the search space may become small enough to allow verification of the corresponding properties even by a lightweight

solver [59]. By this, verification of all properties becomes possible. Recall that this instantiation cannot be done at the design time, because at that point the instantiating values are still unknown. Therefore, the proofs must be rerun if the values of the configuration variables are changed.

3.2.2 The Design Process At Work

In the following, we apply the design flow (Figure 3.2) to the simple example from Section 3.1.

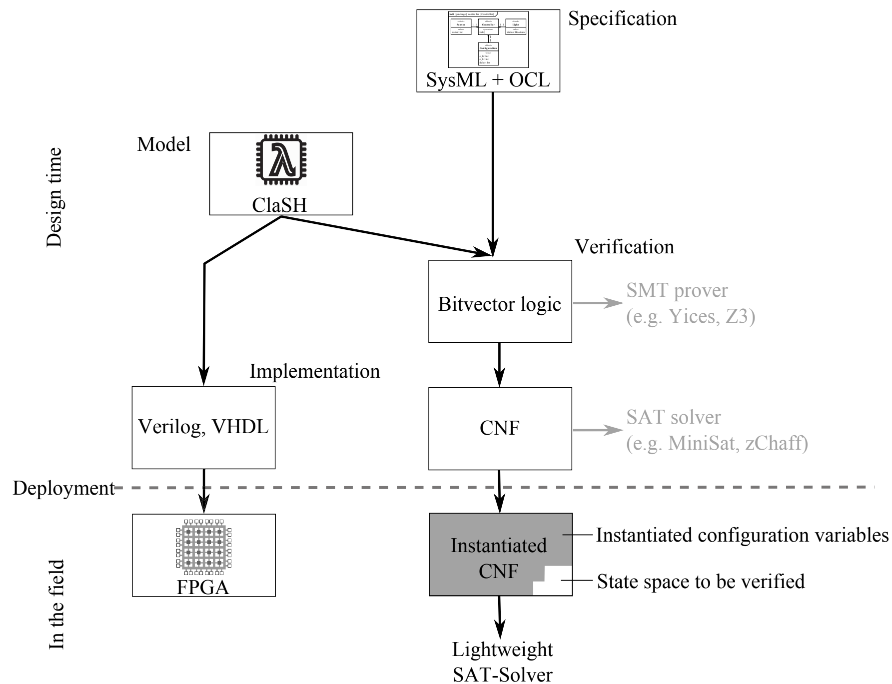


Figure 3.2 Design flow for verification after deployment. We start with modelling the system behaviour, then derive an implementation and verification conditions. By proving the verification conditions, we make sure the system behaves as specified. Due to the large search space, the proofs are not possible pre-deployment. But instantiation of the configuration variables reduces the size of the search space significantly and makes proofs possible post-deployment.

Specification (top of Figure 3.2) A

The specification of the system is provided in terms of a SysML block definition diagram as shown in Figure 3.3. The structure is composed of the controller as the central block, with one luminosity sensor, and one light switch (actuator) connected. The variables specifying the lower and upper threshold of luminosity and the delay when switching off are in a separate block marking them as configuration variables.

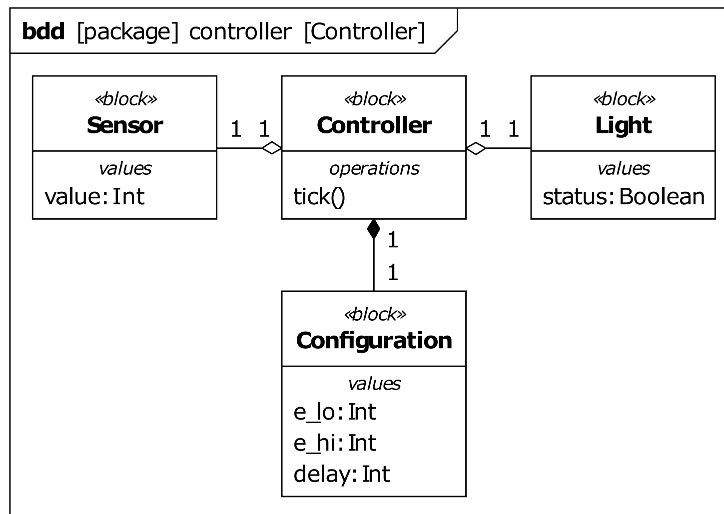


Figure 3.3 SysML specification of the light controller

The behaviour is provided in OCL as shown in Figure 3.4. We model state transitions by an explicit operation `tick()`; The pre- and postcondition of the state transition are denoted as pre- and postconditions of this operation.

Model (middle of Figure 3.2) B

Based on the specification, a Clash model is derived.

Clash is a strongly typed domain-specific language to model hardware. It is embedded into the functional programming language Haskell, and describes the hardware as functions of the language. The

```
1  context Controller
2    def e: sensor.value
3    def off: e > config.e_hi
4    def on: e < config.e_lo
5    def off_s: cnt ⩾ config.delay
6
7  context Controller::tick()
8    post a1: not off implies cnt=0
9    post a2: off implies cnt=cnt@pre+ 1
10   post a3: on implies light.status
11   post a4: off_s implies not light.status
12   post a5: not (on or off_s) implies
13             light.status=light.status@pre
```

Figure 3.4 OCL specification of the behaviour of the light controller

strong type system guarantees that everything we can describe in Clash is still synthesiseable, and allows us to model the hardware at an abstract but still executable level.

The model describes the hardware by combinators (higher-order functions), building up complicated circuits by composing elementary ones. Figure 3.5 shows the model, essentially a finite-state machine (a Mealy automaton) with the luminosity values (*Unsigned 8*) and the configuration as input, the light switch (*Bool*) as output, and an internal state (*ControllerState*) which keeps track of the light switch and a counter to implement the delay when switching off. The function *controllerT* is the state transition function of the automation, taking the state and the input, and returning a tuple of new state and output.

C Implementation (left-hand side of Figure 3.2)

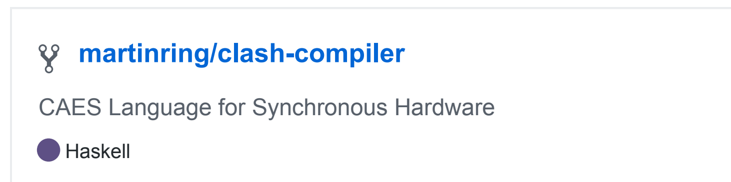
From the Clash model, we generate Verilog, which is then compiled onto the FPGA by the proprietary tool chain of the FPGA vendor (in our case, Xilinx). Thus, the Clash model is the foundation of the verification after deployment.

```haskell
data Configuration = Configuration {
  e_lo :: Unsigned 8,
  e_hi :: Unsigned 8,
  delay :: Unsigned 8
} deriving Show

configurationControllerT :: Configuration
  -> (Bool, Configuration)
  -> (Configuration,Configuration)
configurationControllerT oldConfig (enable,config) =
  if enable then (config,config) else (oldConfig,oldConfig)

configurationController :: Signal (Bool, Configuration)
  -> Signal Configuration
configurationController =
  mealy configurationControllerT (Configuration 63 191 127)

data ControllerState = ControllerState {
  switchState :: Bool,
  cnt :: Unsigned 8
} deriving Show

data ControllerInput = ControllerInput {
  configuration :: Configuration,
  e :: Unsigned 8
} deriving Show

controllerT :: ControllerState
  -> ControllerInput
  -> (ControllerState,Bool)
controllerT
  (ControllerState switchState cnt)
  (ControllerInput (Configuration e_lo e_hi delay) e)
  | e < e_lo                  = (ControllerState True cntn,True)
  | e > e_hi && cnt >= delay = (ControllerState False cntn,False)
  | otherwise                 = (ControllerState switchState cntn,
                                 switchState)
    where cntn = if e > e_hi then if cnt < delay then cnt + 1
                             else cnt else 0

controller :: Signal ControllerInput -> Signal Bool
controller = mealy controllerT (ControllerState False 0)

topEntity :: Signal (Bool,Configuration,Unsigned 8)
  -> Signal Bool
topEntity input =
  controller (fmap (uncurry ControllerInput) $
    bundle (cfgOut,sensor))
  where (enable,config,sensor) = unbundle input
        cfgOut = configurationController (bundle (enable,config))
```

Figure 3.5 Clash model of the light controller

D Verification (right-hand side of Figure 3.2)

To prove the verification conditions, we translate them into CNF, which is suitable as input for reasoning engines such as SAT solvers. This translation proceeds in two steps. We first translate both the Clash model and the specification into bit-vector logic, which in the second step can be translated into CNF by Yices. The translation from Clash is done with an extension of the Clash compiler we have developed for this work:



The translation of OCL is done by the tool-chain described in Chapter 2.

Figure 3.6 shows a small excerpt of the bit-vector representation of the model from Figure 3.5. We are modelling the state transition explicitly, so for each state variable (e.g. *switch*, *cnt*) we have a variable to model the pre-state (here, *preSwitch*, *preCnt*). Figure 3.6 asserts that the state switches to *true* if the luminosity value drops below *e_lo* and it switches to *false* if the luminosity is above the threshold and *cnt* is larger or equal to the configured *delay*.

```
1  (define preSwitch :: bool) ; light switch before
2  (define switch :: bool)    ; light switch after
3  (assert
4    (= switch
5       (ite (bv-lt e e_lo)
6         true
7         (ite (and (bv-gt e e_hi) (bv-ge preCnt delay))
8           false
9           preSwitch
10 ) )  ) )
```

Figure 3.6 Implementation modelled in bit-vector logic (excerpt)

To verify the implementation, we translate the specification from OCL into bit-vector logic; for example, the two clauses *a4* and *a5* from Figure 3.4 become:

```
1 (⇒ off_s (not switch))))
2 (⇒ (not (or on off_s)) (= switch preSwitch))))
```

We generate a CNF formula from the negated conjunction of all five clauses (and the invariants) in Figure 3.4, together with the model from Figure 3.6. This formula is satisfiable iff the specification is violated (because we assert the negated specification).

Because we explore the complete search space (there is no state abstraction involved), this procedure is not only sound but also complete; if we cannot find a counter-example, the verification condition holds for all reachable states within the reduced search space.

Instantiation after Deployment (bottom of Figure 3.2)

E

Finally, the configuration variables are instantiated in order to reduce the search space. This is directly conducted in the obtained CNF. In order to give an impression of the generated CNF, we just consider the very simple assertion $e \leq e_{\mathrm{hi}}$, which translates into bit-vector logic as the assertion:

```
(assert (not (bv-lt e e_hi))).
```

Using only two bits for e and e_{hi}, Yices generates the CNF as shown in Figure 3.7, which represents these bit-vectors as variables, corresponding to the formula. Here, x is an auxiliary variable. e_1 and e_2 denotes the first respectively the second bit of the bit-vector e. The same notation applies to e_{hi}.:

$$
\begin{aligned}
&(\neg e_1 \vee x) \wedge (e_2 \vee \neg e_{hi,2}) \wedge (e_2 \vee x) \wedge \\
&(e_{hi,1} \vee x) \wedge (\neg e_{hi_{,2}} \vee x) \wedge (e_1 \vee \neg e_{hi,1} \vee \neg x).
\end{aligned}
\tag{3.2}
$$

Yices keeps track of the encoding of the variables, i.e. to instantiate the configuration variables, we add corresponding unit clauses, e.g. to

```
1  c   e_hi ⟶ [5 6]
2  c   e ⟶ [3 4]
3  p cnf 7 6
4  -3 7 0
5  4 -6 0
6  4 7 0
7  5 7 0
8  -6 7 0
9  3 -5 -7 0
```

Figure 3.7 CNF in DIMACS format for a very simple assertion. Lines starting with c are comments; the line starting with p states the number of variables and clauses; the following lines are the clauses, each line containing one conjunct consisting of a disjunction of variable i or its negation $-i$ (terminated by 0). A suitable representation of this format is used post-deployment.

instantiate e_{hi} with the value 2 (10 in binary) we add two unit clauses stating

$$e_{hi,1} \wedge \neg e_{hi,2}.$$

The instantiations now significantly reduce the search space. This can be exploited to solve the resulting instance after deployment using a lightweight solver. As we can see, the actual reduction of the search space depends on the values we instantiate the variables with.

3.3 Evaluation

So far, the proposed methodology has been illustrated by means of an intentionally rather limited example. Moving on from that, we have applied the idea of verification after deployment, and the proposed verification as described in Section 3.2, to more sophisticated home automation controller in order to demonstrate its applicability.

The home controller has been realised on top of a ZedBoard, which comprises an ARMv7 core running Linux to control a Xilinx FPGA, and which for the purposes of verification has been equipped with a lightweight SAT solver [59]. The obtained results are summarised in this section. Furthermore, we also discuss possible ramifications

which have to be considered when utilizing the proposed methodology in practice.

Evaluation 3.3.1

The proposed methodology has been evaluated on a set of systems which are natural extensions of the light controller considered above to highly versatile home automation controllers as follows:

simple
: The simple light controller with one light and one luminosity sensor (as considered in the running example).

average
: An extended version of the controller which includes up to 16 sensors to be connected and controls one actuator by averaging the values obtained by those sensors. Input and output are generic, i.e. we can control any kind of actuator and read from any kind of sensor as long as it gives us integer values.

weighted_avg
: A similar version with 32 sensors that allows to add a configurable weight to each sensor when computing the average.

smart
: A smart home controller, which allows up to 32 sensor inputs to be connected to up to 32 actuator outputs. Each input can be connected with each output, making the controller very versatile and resulting in a huge search space. The smart home controller can be used e.g. to control lights, heating and blinds for a number of rooms in an office setting.

multiplier
: A 16 bit multiplier component, used to apply the weights in *weighted_avg* and *smart*. Can be verified with a constant factor once the configuration is set.

For all these systems, we have specified their intended behaviour in OCL, similar to the specification of the simple light controller in Figure 3.4, and have verified that the implementation satisfies this specification.

Table 3.1 and Table 3.2 list the results. Column *System* gives the name of the considered system. The remaining columns summarise the results in two groups: Table 3.1 for verification according to the established verification flow (i.e. verifying all properties at design time) and Table 3.2 for the verification methodology proposed here (using the lightweight solver on the target system). For each group, we give the size of the search space (i.e. the number of possible solutions to be checked); the number of variables; the number of clauses of the resulting CNF; and the run time (in seconds). The run time is measured on systems which would typically be used for verification, so they are directly comparable: for the established verification flow, a compute server (Intel Xeon E3-1270 v3, eight cores, 16 GB memory) and, for the proposed verification flow, the ZedBoard (ARMv7, 1GB memory).

Table 3.1 Evaluation Results (Established Flow)

System	Search space	Variables	Clauses	Time
simple	2^{41}	161	539	$< 0.1s$
average	2^{177}	11807	40086	$131.0s$
weighted_avg	2^{545}	43569	146642	$> 24h$
smart	2^{9504}	1421153	4761633	$> 24h$
multiplier	2^{32}	1177	6096	$> 24h$

The obtained results clearly show the benefits of the proposed approach. Typical embedded systems (as the ones considered here) allow for a huge variety of configurations. As shown in Table 3.1, this results in a rather large search space and SAT instance for the verification, which takes a significant amount of time to solve (in some cas-

Table 3.2 Evaluation Results (Proposed Flow, with randomly assigned values)

System	Search space	Variables	Clauses	Time
simple	2^{17}	131	539	$< 0.1s$
average	2^{137}	8181	40086	$1.4s$
weighted_avg	2^{265}	31374	146642	$28.5s$
smart	2^{544}	1421153	2704606	$1.5s$
multiplier	2^{16}	809	2467	$418.0s$

es, the corresponding verification task could not be solved within the given time-limit of one day). In contrast, after deployment, configuration variables can be instantiated with their actual values, as discussed in Section 3.1. This substantially reduces the search space and allows to solve the verification task even on the limited resources of an embedded system as shown in Table 3.2.

Of course, the search space is only one complexity indicator: as the multiplier system shows, even a comparatively small search space may require a long time to be verified, because of its inherent complexity. However, the proposed verification flow reduces the run time significantly in this example as well, and thus allows us to verify a system which was previously out of reach for established tools.

Practical Exploitation 3.3.2

Our approach may be applied in various ways. In the following we illustrate a possible practical application to the design of a smart home controller as described above.

Requirements and properties are established during design time and checked with contemporary verification tools. Refinements are tracked and verified down to the electronic system level. All properties which cannot be automatically checked during design time are

then collected. Some of these properties might be provable with interactive theorem provers. The effort has to be weighted up with the win here. Those properties which cannot economically be proven are then prepared for self-verification using our approach.

In the deployed system, a verification controller is constantly watching the values of the configuration variables and triggers a proof if a value change is requested. For example, if a light is connected to the smart home controller, the configuration is updated, and the proofs have to be re-run. Since the system would now be in an unverified state, it will either stop operating or defer the value change until the proofs have successfully finished; this way, it continues operating with guaranteed safety. (If the risk is considered acceptable, the system might instantly change the value and continue to operate while the proofs are running.)

This results in a transient state where the system is unverified for the time it takes to conduct the proof. There are three possible ways to cope with this:

(1) For acceptable risks, the system can just continue operating while the verification is running in parallel.
(2) We can delay the change of the variable and ignore the connected light until correctness has been proven. This sacrifices function for safety.
(3) We can stop operation and only continue after the system is proven safe again. This potentially violates non-functional requirements on timing but safety and function are unaffected.

None of these situations is desirable and thus the verification controller might use statistical observations for the prediction of future variable-states and proactively verify them during idle-time. If any of these states occurs, the system can instantaneously continue operating with guaranteed safety (see also Section 6.1).

If a proof fails for the resulting configuration, the system informs the user about the failed proof. The user can disconnect the sensor again or try a different configuration until the proof succeeds and the change results in a safe state. This especially means that the system can still operate safely even though some functionality is missing. Furthermore, the manufacturer is informed about the failed configuration, and can use this information to take appropriate measures.

Discussion 3.4

The results obtained by the conducted cases studies summarised above clearly show the promises of the proposed verification methodology. However, some obvious ramifications have to be discussed when evaluating the general applicability of this methodology.

The proposed methodology obviously requires the system to be equipped with on-board verification tools to conduct the verification tasks. Since the considered systems are substantially less powerful than usual desktop systems or verification servers, this requires lightweight but still efficient versions of those tools. Here, recent developments on lightweight methods [59], [60] as well as endeavours towards efficient hardware solvers [61], [62] provide promising platforms for this purpose. Besides, it should be noted that the proposed verification methodology yields an exponential reduction in the search space, so even less powerful verification tools might be able to cope. Our evaluation results corroborate this assumption.

Besides that, the obvious question is what happens if the verification after deployment fails, and the deployed system turns out to be erroneous? This would be rather unfortunate, but we still believe that conducting verification after deployment has its value. First, note that this is a strict improvement over the existing situation, where the error would not have been detected at all, while verification after deployment at least shows its existence. This gives vendors the possi-

bility to react (e.g. by calling the system back, issuing software patches or hardware fixes). Second, verification failure does not necessarily indicate an erroneous system; it may equally indicate that the configuration variables are instantiated with erroneous values. In this case, the system may just pause until it is re-configured with allowed values, in this way guaranteeing correct functionality.

Our approach differs from *run-time verification*, which is concerned with "checking whether a *run* of a system under scrutiny satisfies or violates a given correctness property" [63]. The central notion of run-time verification is the trace (or run) of a system, and central questions are how to derive monitors checking a concrete run against an abstract specification. The logics employed are typically temporal or modal logics. In our work, we are not concerned with monitoring the system at all, we instead *specialise* given variables in an abstract specifications if they do not change often.

3.5 Conclusion

This Chapter introduced a general approach to Self-Verifying Systems and showed it's feasibility by applying it to several case studies. We were able to show the general applicability of the approach but a couple of important questions have yet to be answered:

(1) When exactly should the verification take place and what are the consequences of late vs. early verification?
(2) Which parts of a system should belong to it's *configuration* and how can we systematically determine these parts?

The following chapters will address these questions respectively.

4 Design of Self-Verifying Systems

In this chapter, we investigate the effects of self-verification on the development. That is, we want to explore *when* to prove properties and which ones, and we want to investigate how self-verification interacts with the development process. For this, we introduce a more abstract view on self-verification and widen the somewhat simplistic concept of *configurations* to *trigger transitions*: Transitions of a system that trigger a verification (i.e. in terms of Chapter 3, a change of the configuration).

4.1 Self-Verification, Design Time & Run-time Verification

The key advantage of self-verification as described in Chapter 3 is that after deployment, the concrete values of parameters may become known for verification. Some may be instantiated early on after deployment, and not change after that at all, or only very infrequently; others may change, but not that often; and even others may be sensor data which are read in small intervals, but where the rate of

change may be limited. All of this information may be utilised at run-time for more efficient verification.

This observation hinges on the fact that proving a property ϕ depends, *inter alia*, on the number of free variables in ϕ, and that parameters as mentioned above usually occur as free (or universally quantified) variables in ϕ. Then, proving $\phi\left[\begin{smallmatrix}t\\x\end{smallmatrix}\right]$ with a ground term t instantiated for x is typically orders of magnitude easier than proving ϕ.

Comparing self-verification to run-time and *a priori* design time verification on a more abstract level, we consider specific runs of the system $\langle\sigma_i\rangle_{i\in\mathbb{N}}$, consisting of states σ_i, and a safety property ϕ. Usual design time verification proves the general property that for all runs, $\forall i.\ \phi(\sigma_i)$, i.e. the safety property holds for all states. In OCL and related formalisms, this is achieved by an inductive argument, showing that we start in a safe state, $\phi(\sigma_0)$, and that from a safe state we can only get to a safe state, $\phi(\sigma_i)$ implies $\phi(\sigma_{i+1})$.

Run-time verification, on the other hand, considers whether a specific run satisfies $\forall i.\ \phi(\sigma_i)$ and does not restrict the transitions of the system; unsafe states can be reached, but this is always detected.

In self-verification, instead of restricting transitions, we classify them into trigger transitions and ordinary transitions. The idea is that when the system goes through a trigger transition $\sigma_i \to \sigma_{i+1}$, self-verification shows that all states σ_k reachable with ordinary transitions from σ_{i+1} are safe, i.e. $\phi(\sigma_k)$. If another trigger transition is reached, the self-verification is run again. Note that the classification of trigger transitions and ordinary transitions depends on the particular ϕ and is a design decision (see Section 4.2 below). *A priori* and run-time verification can be seen as extreme cases of self-verification: in design time verification only one transition (the one leading to the initial state of the system) is classified as a trigger transition, while in run-time verification every transition is a trigger tran-

sition. Figure 4.1 illustrates the effect of different sets of trigger transitions for one system.

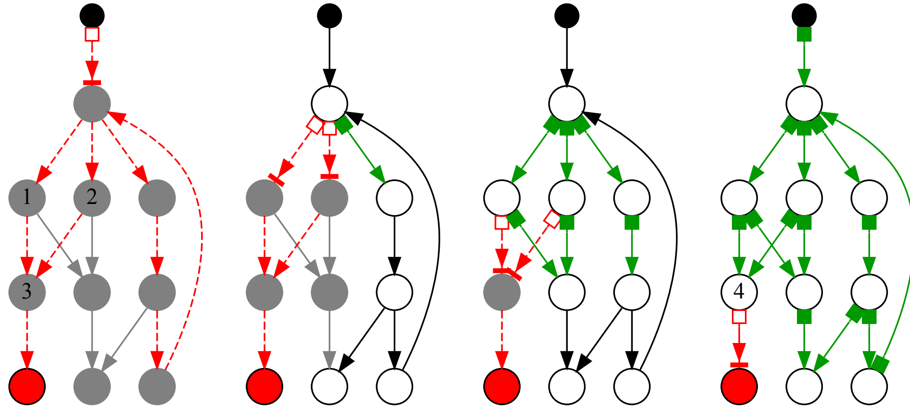


Figure 4.1 Four different points in time chosen for verification, from design time (leftmost) to run-time (rightmost). Trigger transitions are marked with small boxes; they trigger verification tasks which show that every possible path through the state space which does not include other trigger transitions is safe. Green boxes mark successful verification, and red boxes mark failed verification tasks. The solid red state is unsafe; it violates the safety property ϕ. Grayed-out states are not reachable, because they come after a failed verification (open red box). Design time verification (on the left) would identify the system as erroneous and prohibit its execution. Second to left, the system is verified early after deployment and thus is allowed to execute only a small fraction (6 transitions) of the system, blocking two transitions and leaving 6 transitions unreachable. Third to left, most of the system is executable (11 transitions) but two transitions are blocked and one transition is not reachable. The rightmost example allows all but one transition. Note that in the last example the system gets deadlocked in state 4 when taking the leftmost path.

Because the effort to state and prove ϕ increases with the number of states we want to cover, self-verification allows us to strike a balance: we may prove ϕ with little effort for a small number of states, and so have to reprove it more often, or we may prove ϕ for more states, but with more effort.

When we specify the desired behaviour of the system with design time verification, we need to state the required preconditions very

precisely — they need to be strong enough to be able to actually show that the system globally satisfies the specified properties, and to preclude unwanted behaviour, but weak enough to still allow all desired implementations. If we move verification into run-time, we can relax preconditions at design time, allowing for more readable specifications and speeding up the development process. Consider Figure 4.1 again: to make the system usable as well as correct, one would have to e.g. refine the specification (or the implementation) to exclude the transitions from states 1 and 2 to 3. With self-verification, we can allow a more liberal specification or implementation and still remain safe, making the development process easier.

Thus, in essence specification becomes easier and faster to write, and moreover we are liberated from having to prove everything *a priori* and can instead adapt the proving strategy to the problem at hand.

4.2 Case Study

In the following, we introduce a case study building loosely on Abrial [28]. The case study is simple enough to be easily understood, yet complex enough to show the subtle effects of verification at different points in time. Note that this case study is strictly different from the similar example in Chapter 2 and must not be confused.

4.2.1 Informal Description

To motivate our case study, think of a building where fine-grained access control is needed for security or safety reasons, e.g. a nuclear power plant, but which also needs to be able to be evacuated very fast in the case of an emergency. In that case, we want to be able to eliminate access control (to allow fast evacuation) and just open some of the doors in such a way that all users are able to get out, but no user gains access to a room where they are not allowed to enter.

More precisely, we have a *building* consisting of several *rooms*. The rooms are connected by *doors*, which are unidirectional (think of turnstiles; normal two-way doors are an obvious generalization). Thus, doors lead from one room to another one, which is equivalent to each room having a set of entries and exits.

Users are represented in the system by *cards* which regulate the access to rooms. (In the following, we use cards and users interchangeably; the formal specification only has cards.) Each card authorises access to a set of rooms, by restricting passage through the doors. The access control system operates in two modes: in normal mode, a door may only be passed (using a card) if the card authorises access to the room the door is leading to. However, we can declare an emergency for the whole building; in that modus, some doors are opened, allowing anyone to pass through.

Opening doors in an emergency is subject to two *safety properties*: firstly, it should allow any user (card) to eventually arrive in a safe room, and secondly, it should not allow any user to enter a room they are not authorised to. A subset of rooms is considered to be *safe*; in the simplest case, this can just be the outside modelled as a room.

As an example for the necessity of the safety properties, take the nuclear power plant: even in case of an emergency, one would not want anybody to exit through the reactor core.

This rather innocuous specification allows some subtle effects. Consider the simple building in Figure 4.2. The depicted situation violates the safety property, as in case of an emergency, we cannot disable access control and open the doors in such a fashion that neither user A or user B are allowed to access rooms they are not authorised to (rooms b and a, respectively), and at the same time both are able to get to a safe room (s).

Hence, we need to prevent a situation like this from happening. This could be done by

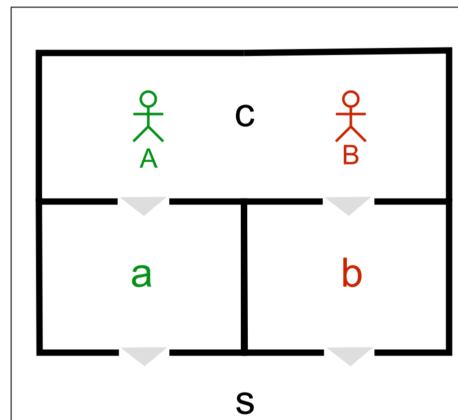


Figure 4.2 Example of a very simple building. The user with card A is authorised for room a, user B is authorised for room b, both are authorised for rooms c and s. Room s is the only safe room (it is the outside). The situation shown violates the safety property.

- either restricting the layout of the building in such a way that situations like this do not happen (this is what is usually done, with layouts were corridors are the default escape route, and users do not have to traverse long sequences of rooms);
- or by restricting the authorizations of the cards in such a way that a situation like above does not happen;
- or by checking that *before* a users enters a room no situation violating the safety property like above is created.

Formal Specification 4.2.2

We can now give a formal specification of our access control system. We will use the subset of SysML and OCL introduced in Chapter 2, where block definition diagrams model the structure of the system, and OCL constrains the dynamic behaviour.

In Figure 4.3, we can see blocks modelling the building, doors, rooms and cards respectively. The building has a Boolean attribute *emergency*. A door leads from exactly one to another room, but a room may have many (or no) entries and exits. A door may only connect rooms which are part of the same building:

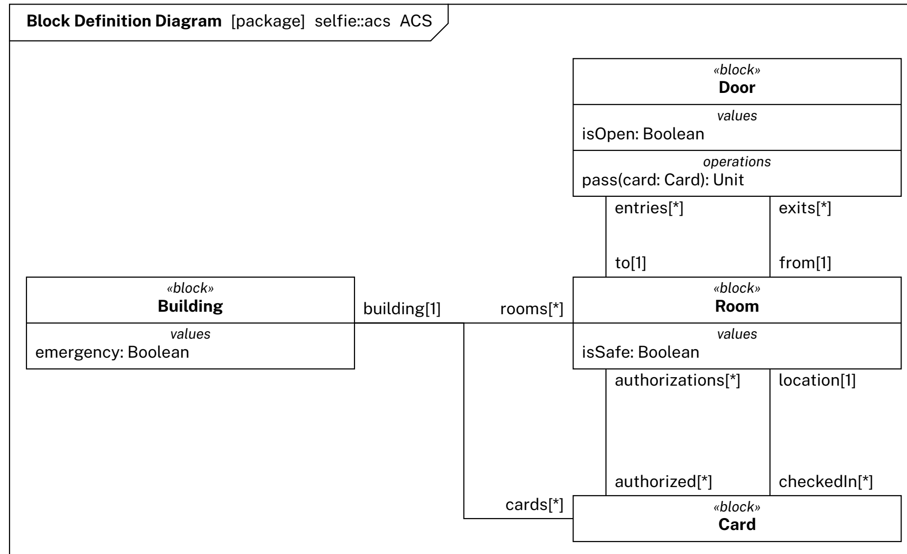


Figure 4.3 Formal specification of an access control system.

```
1 context Door
2   inv: from.building = to.building
```

Furthermore cards are also associated to buildings and may only authorise access to rooms which belong to the same building:

```
3 context Card
4   inv: authorizations→forall(r| r.building = self.building)
```

Cards have a set of *authorizations* (rooms which the holder of the card is allowed to enter) and exactly one *location*, which determines the current location of the card, and which must always be contained in the set of authorizations[1]. On the other hand, rooms have a set of *authorised* cards (those cards which have the room in their set of authorizations), and a set of *checkedIn* cards (the set of cards whose location is this room).

```
5 context Room
6   inv: checkedIn→forall(p | authorized→contains(p))

7  context Card
8   inv: location→forall(r | authorizations→contains(r))
```

1. We assume an idealised scenario where we can reliably track the location of a card holder and can prevent that people share their cards

Rooms have a Boolean attribute *isSafe* which determines whether the room is safe during an emergency. A door has a method *pass*, which determines whether a given card is allowed to pass. This is the case if either the door is open (see immediately below), or if the card is in the room this door is opening from, and the card is authorised for the room the door is opening to. We have encapsulated this precondition as an OCL function *mayPass* in order to reuse it later. The postcondition of the *pass* method is that the *location* of the card has changed to the room the door is opening to. Doors are only allowed to be opened in case of an emergency.

```
 9  context Door
10    def: mayPass(card: Card): Boolean =
11      isOpen or from.building.emergency
12      and card.authorizations→contains(to)
13    inv: isOpen implies from.building.emergency

14  context Door::pass(card: Card):
15    pre: mayPass(card) and card.location = from
16    post: card.location = to
```

We now want to the safety property: in an emergency, users can always reach a safe room, yet no user has access to a room they are not authorised to. To formalise a user being able to reach a room, we formalise the notion of recursive *access*, which models the traversal along a sequence of connected rooms: users have access to the room they are currently in, and recursively to all rooms which can be reached through doors which may be passed (i.e. rooms which have an entry from an accessible room that this card has access to). We formulate this notion as an OCL function *hasAccess* which for a given room determines whether a given card has access to this room. Since OCL does not allow non-terminating functions we pass the set of already traversed rooms to the helper function *hasAccess$* such that we do not traverse cycles:

```
17  context Room
18    def: hasAccess(card: Card): Boolean = hasAccess$(card,Set{})
19    def: hasAccess$(card: Card, visited: Set(Room)): Boolean =
20      card.location = self or
21      visited.excludes(self) and entries→exists(e |
22        e.mayPass(card) and
23        e.from.hasAccess$(card, visited→including(self)))
```

We can now specify the safety properties: firstly, that users can always reach a safe room, and secondly, that users only have access to rooms they are authorised for:

```
24  context Card:
25    inv safe1: building.rooms→exists(r |
26      r.isSafe and r.hasAccess(self))
27    inv safe2: building.rooms→forall(r |
28      not r.authorized→contains(self) implies not r.hasAccess(self)))
```

4.2.3 When to Verify

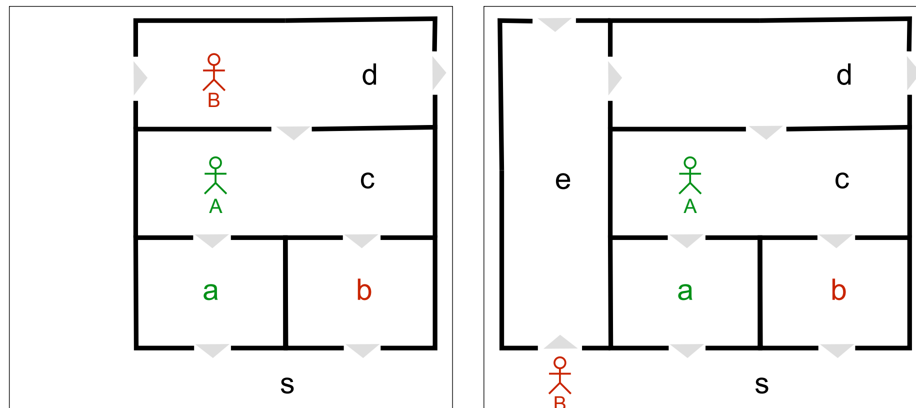


Figure 4.4 Situations which are safe. On the left, user B cannot enter room c until user A has left. On the right, a similar situation, but the user B may have taken the long path through room e and d quite unnecessarily before not being able to proceed further.

In order to preclude an unsafe situation as in Figure 4.2, we have to show our system satisfies the safety property. Of course, in full generality — universally quantified over all buildings and all authorizations — the safety property does not hold; we can easily find counterexamples (such as Figure 4.2). If we want to show the safety prop-

erty at design time, we have to formalise conditions which are sufficient for the safety property (i.e. preclude unsafe buildings).

With self-verification, we can show the safety property after deployment, at different points in time:

(a) right after deployment to a specific building, for all possible cards, authorizations and allocations of users to rooms; or
(b) after authorization has changed, for a specific building, but for all possible allocations of users to rooms; or
(c) when a user requests access to a different room: if the new configuration of the user in this different room is unsafe, access is not granted.

In case (a), we would either need an explicit and sufficient characterization of "every user always has a safe exit route", or we need to search a lot of instances (all paths for all users from all rooms). For most buildings, we will be able to find counterexamples of unsafe configurations of users and access rights, but we may be able to restrict access rights in such a way that we can prove the safety property. If we can prove the safety property at this point, we are done, but this may not always be possible.

The other extreme case is (c); this is fairly straightforward to verify but might be inconvenient to the user. (Thus, this is an example of making a system safe by restricting its availability.) Consider the situation in Figure 4.4 with the same authorizations as in Figure 4.2. On the left, user B cannot enter room c until user A has left, because otherwise we would have the situation from Figure 4.2 which is not safe. This might result in situations like on the right of Figure 4.4, where user B might take a long tour through room e to room d only to find they cannot proceed any further.

A good compromise is case (b): we verify the safety property each time the authorizations change, for a specific building and specific authorizations. In most cases, this should be reasonably efficient —

the search space is through all possible allocations of users to rooms — but still precludes unsafe allocations.

Note how self-verification allows us to relax the development process: because we can prove the safety property at run-time, we do not need to specify all its preconditions at design time (here, we do not need to characterise the preconditions to make buildings and authorizations safe). This makes the development process more *agile* without compromising safety.

4.3 Realization

4.3.1 Applying the Design-Flow for Self-Verification

We apply the design flow introduced in Chapter 3 to our case study. As demonstrated in Section 4.2, we use the same subset of SysML[2] together with OCL as a specification formalism. Block definition diagrams and state machine diagrams can be given a formal semantics (which is not the case for all SysML diagrams), so our specifications have a mathematically well-defined, formal meaning. This is indispensable if we want to perform formal correctness proofs.

We use our textual representation of block definition diagrams and state machine diagrams (Figure 4.5). Parts of the corresponding OCL specifications have been shown in Section 4.2 above.

The implementation is given as an executable *system model*. To stay independent of a specific programming language, we again use the functional hardware description language Clash [26] as modelling language, since it allows us to simulate the system as well as synthesise an implementation in VHDL or VeriLog. Another possibility with more commercial traction would be SystemC, but that has a less clear semantics and it is embedded in C++, technically a lot more awkward to handle (in Clash, adding proof support was merely a question of

2. The case study only uses block definition diagrams.

```
bdd [package] selfie::acs [ACS]
-------------------------------

block Building
  references
    rooms: Room[*] ← building
    cards: Card[*] ← building
  values
    emergency: Boolean

block Door
  references
    from: Room[1] ← exits
    to: Room[1] ← entries
  values
    isOpen: Boolean
  operations
    pass(card: Card)
      pre: mayPass(card) and card.location = from
      post: card.location = to
  constraints
    def: mayPass(card: Card): Boolean =
           if from.building.emergency then isOpen
           else card.authorizations→contains(to)
    inv: isOpen implies from.building.emergency

block Card
  references
    authorizations: Room[*] ← authorized
    location: Room[1] { subsets authorizations } ← checkedIn
    building: Building[1] ← cards
  constraints
    inv: authorizations→forall(r| r.building = self.building)
    inv: building.rooms→forall(r| not r.authorized→contains(self)
                                  implies not r.hasAccess(self)))
    inv: building.rooms→exists(r| r.isSafe and r.hasAccess(self))

block Room
  references
    building: Building[1] ← rooms
    exits: Door[*] ← from
    entries: Door[*] ← to
    authorized: Card[*] ← authorizations
    checkedIn: Card[*] { subsets authorized } ← location
  values
    isSafe: Boolean
  constraints
    def: hasAccess(card: Card): Boolean =
           card.location = self or entries→exists(e |
             e.mayPass(card) and e.from.hasAccess(card))
```

Figure 4.5 SysML block definition diagram expressed in SPECifIC SysML

adding an additional backend; in SystemC, we do not even have an explicit representation of the model to start from).

Our tool chain (as introduced in Chapter 2) reads the SysML and OCL specification, performs the appropriate type checks, reads the Clash model, and generates the corresponding first-order proof obligations in bitvector format (first-order logic with limited width integers as datatypes). The proof obligations are essentially obtained by taking a representation of the system model in bitvector logic, and showing they satisfy the OCL constraints (pre/postconditions and invariants). They can be either processed at design time by an SMT prover such as Yices or Z3, or transferred to run-time. Proving at run-time is either performed by an SMT prover running on the target system, if the latter is powerful enough, or by converting the proof obligations into conjunctive normal form (e.g. using the Yices prover) before transferring it to the target system, and using a SAT solver at run-time (either as a lightweight software SAT solver [59] or even a hardware SAT solver [62])

4.3.2 The Demonstrator

If we implement the case study in our usual design flow, we derive a hardware implementation, e.g. on an FPGA. In order to explore the implications of proving at different points in time, and to demonstrate the effects of self-verification in an easily accessible setting, we implemented the case study as an interactive demonstrator. For this, we leveraged the two previously developed tool flows for the instantiation of our model from Chapter 2. This allows us to semi-automatically translate the above SysML definition into an executable system by generating implementation stubs as well as translating the model into a refinable SMT instance.

Simulating the hardware turned out to be very slow, so instead we chose to adapt our flow: the implementation is an interactive SVG, with the dynamic behaviour implemented in TypeScript.

The core of the system is generated as implementation stubs, using an adapted form of our design flow (see Figure 4.6).

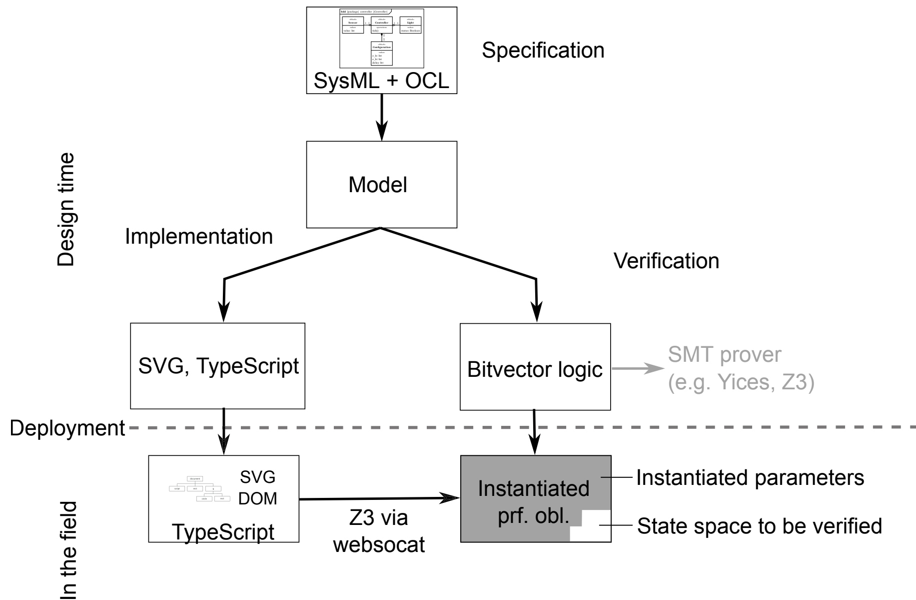


Figure 4.6 Design flow adapted to our demonstrator.

We have chosen TypeScript [64] as the target language (TypeScript is like JavaScript, but with added type security), because it allows us to dynamically modify the abstract syntax tree (the DOM) of the SVG. This allows the demonstrator to be displayed and run on any recent web browser. In addition to the specified behaviour we manually implemented means to add and remove cards and change their access rights, and reading building topologies from a non-interactive SVG. We have implemented access cards (and implicitly their owners) as automated agents which randomly roam the building. This allows us to observe the implications of the different points in time of the verification; for example, the behaviours mentioned for case (c) in Section 4.2 above manifest themselves in agents hovering in one place unable to proceed because of the violation of the safety property this would incur.

The generated SMT proof obligations can be processed by an SMT prover at design time. As mentioned above, the prover quickly finds

counter examples since our specification can easily be violated in general. By adding run-time information in the form of assertions, we refine the instance on the fly. This was realised by establishing a WebSocket connection between the SVG and the Z3 prover. For this, we use the *websocat* utility, which wraps a WebSocket server around a command-line program. This allows us to load the general proof and then incrementally send assertions restricting the state space.

Technically, the arbitrarily mutable state of our simulation is in principle not compatible with the monotonous nature of adding assertions: assertions can only add information but not change or remove. Fortunately, SMT-LIB (the common language used by most SMT provers) allows us to use scopes (with the commands *push* and *pop*) for this. In order for this to work, we introduce a fixed order in which information is added, which is based on the order of execution in the system, ideally corresponding to the frequency of change. First, we add the general building topology, then the access rights, and after that, the tracked locations of the card holders. Between every assertion, we save the current size of the assertion stack with the *push* command. If any information changes, we remove the assertion with the now outdated information as well as any assertion which came afterwards. Then we only need to add the updated assertions. Depending on the point in time chosen, we can check satisfiability anywhere between.

An interesting feature of our implementation is that we did not implement any algorithm which opens the doors. Instead, we use the prover to give us a model of the existentially quantified safety property, which states that there must be a safe way to exit (i.e. a set of doors to open in case of emergency). Through self-verification not only did we not have to characterise buildings, access rights or safe paths through the building, we even did not have to implement a path finding algorithm at all.

The demonstrator is shown in Figure 4.7. It connects the implementation to the proof engine running the SMT instance. We can manually choose one of the three different information levels for the proof, which result in different assertions being added as well as different triggers for the proof.

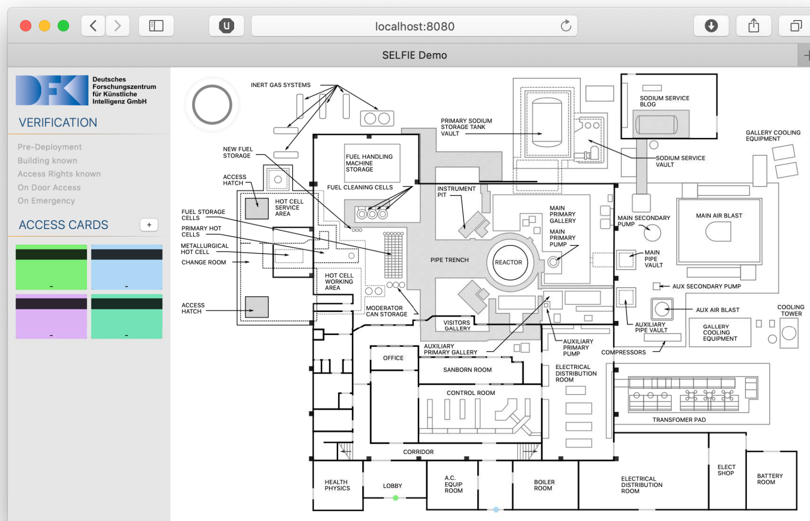
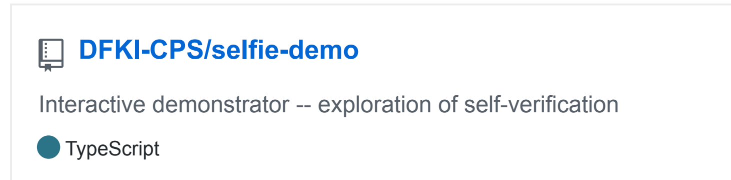


Figure 4.7 The demonstrator is implemented as an interactive SVG document, displayed here in a web browser.

Users can explore the consequences of the different points in time for the self-verification. For example, if they choose to verify early on (after a new card has been added or access rights change) and add a lot of cards, they will notice a considerable slow-down when adding new cards or changing access rights. If they choose to verify late (before a user enters a room), and construct situations like in Figure 4.4, they will realise how users congregate in front of a room unable to get in. (The demonstrator is intended to be used together with additional interactive explanation, not stand-alone, as situations like this will have to be constructed consciously.)

The source code of the demonstrator is publicly available on GitHub:



4.4 When to Prove

The focus of the case study has been to investigate the implications and consequences of the point in time at which the proof of safety properties take place at run-time.

Generally, the earlier we can prove, the more general the proven safety property, but the larger the search space is and subsequently the longer it will take. How to pick the right points in time depends on the actual system and is very much a design decision.

However, we have made a number of observations which can help to assist in finding the right set of trigger transitions. The set of trigger transitions should be large enough such that verification tasks can be completed in a timely manner (again, acceptable verification times depend on the concrete use case) but reduced in a way such that no critical transition is included. Trigger transitions might be prohibited by self-verification in case the specification is violated (fails to verify in the concrete instance), so critical transitions should not be included in the set of trigger transitions: e.g. if we verify the existence of an escape route in case of an emergency it is clearly too late to handle failure. On the other hand, administrative operations like changing access rights are far better suited to be included as trigger transitions, since a potential failure is presented to a trained user of the system. Lastly, one should avoid transient states (e.g. a user is inside a security gate) which can only be left through trigger transitions since self-verification may lead to a system dead-locked there, as in Figure 4.1.

Conclusion 4.5

The vehicle of our investigations in this chapter was a case study consisting of an access control system, which is parameterised in many dimensions (the building under control, the access rights, the users) that can be instantiated at different points in time.

In order to make our results concrete and tangible, we have developed a demonstrator — the access control system implemented as an interactive SVG, which can be viewed and run in any web browser. Users can directly experience the effect of choosing different verification triggers.

The demonstrator also exhibits the applicability of self-verification and the versatility of our tool chain, which could be adapted to support a different implementation platform (SVG and TypeScript instead of Clash) with moderate effort.

This raises the question of the general applicability of the approach. As presented here, some kinds of safety-critical systems could not be addressed adequately, namely fail-safe systems [65], where there is no default safe state which we can always revert to if self-verification does not succeed. On the other hand, an attractive avenue for further exploration is "just-in-time verification", where one tries to prove properties at run-time as they are needed (see Section 6.2)

In Chapter 5, we will further investigate how the designer can be assisted in the design decisions; in particular, how we can systematically find out which variables offer the most reduction in proof time when instantiated.

5 Proof Partitioning

The central means to reduce the search space during run-time and, by this, reduce the run time of the reasoning engine is to set a certain amount of the given variables to a fixed value. While the general methodology has been explored in the previous chapters, the question which variables to fix in order to achieve the largest reduction of verification run time has not been addressed yet. While in theory fixing one Boolean variable would reduce the search space and run time by half, actual instances show a much smaller and less uniform reduction due to the optimizations by the proof engine. Some variables may hardly have an effect at all, while others may immediately cut down a day-long verification process to a few moments. Because of that, it is essential for verification engineers to have a clear understanding about the impact of fixing a particular variable on the verification run time, so they can follow the general idea of fixing some variables in order to get a partial result out of the verification process covering as many cases as possible. However, no systematic investigation on this effect has been conducted so far.

In this chapter, we introduce a methodology to analyze verification run time, and to measure it practically in a meaningful way. The main problem is *how many* and *which* variables are fixed. For this, we first state a formal criterion describing an optimal solution to this problem. Based on that, a cost function is defined which can be used to

employ stochastic and heuristic methods in order to eventually determine solutions optimised for this goal.

Using a proof-of-concept implementation based on evolutionary algorithms, we were able to confirm the potential of the proposed methodology. In fact, experimental evaluations confirmed that this methodology indeed determines a set of variables to be fixed which keeps the verification run time within specified limits while still covering as much as possible of the search space.

In general, the methodology works for any other heuristic which optimises with respect to a given cost function, and the proposed analysis method is independent from both the reasoning engine and the underlying logical language, i.e. we treat the reasoning engine as a completely opaque black box which either proves a proposition or not.

This offers valuable information for designers following the approach of the previous chapters, to choose which parts of a proof offer the highest potential to be postponed for self-verification.

5.1 Fixing Free Variables

In the following, we consider a verification problem as a single proposition[3] ϕ that shall be proven with contemporary reasoning engines such as SAT solvers [13], [66], SMT solvers [15], [17], [67], or similar. The particular logic and reasoning engine used do not matter, as long as the proof procedure is fully automatic. We are interested in problems that cannot be solved using the given resources, where the

3. Note that a number of verification conditions can of course always be combined into a single proposition by conjoining them. Furthermore, we consider all variables to be Boolean. This does not restrict the methodology (because other types such as integer variables can be encoded as bit vectors) but significantly simplifies the exposition in the following.

verification process would be aborted and the verification engineers would get no result at all.

In contrast, when enough variables are set to fixed values (we say the variables are *fixed*), the search space is reduced and the reasoning engine eventually yields a verification result. Even if such a result would not cover all instances of the verification problem, proving an instance of ϕ may still be of potential value.

This yields the questions *how many* and *which* variables should be fixed. So far, no detailed analysis exists on whether the number and selection of variables matters, on by how much the verification time is actually reduced, and how to measure these effects in the first place. In an idealised scenario, answers to these questions would be as sketched in the following example:

Example 5.1

Consider a verification problem ϕ whose complete verification takes a certain time T_ϕ. Setting *all* variables of ϕ to a fixed value will allow for a more or less instantaneous completion of the verification task.[4] Moreover, in an idealised scenario, the proof time would be reduced exponentially with respect to the number of fixed variables. This is sketched by the *green solid line* in Figure 5.1, showing an idealised graph plotting the (presumed) average proof time (in logarithmic scale) over the number of fixed variables. In this idealised scenario, answers to the two questions raised above are trivial: It does not matter *which* variables are fixed (any differences are averaged out) and the *number* is basically determined by the available resources, i.e. the available time (on the y-axis) determines the corresponding number of variables (on the x-axis).

4. In some logics (e.g. with nested quantifiers), this might not be the case, but the general principle that proving ground term propositions is much faster is still valid.

However, such an idealised scenario almost never occurs. In fact, it quickly becomes clear that the relation between the number of fixed variables and the proof time is rather erratic. Again, this is illustrated by means of an example:

Example 5.2

Consider a representative benchmark taken from the SMT-LIB benchmark library [68][5] for which the relation between proof time and sets of fixed variables have been evaluated. The obtained results are shown in Figure 5.1. Here, each data point at (n, t) corresponds to the average proof time t of ϕ with n different variables fixed. As can clearly be seen, there is no obvious relation between proof time and the number of fixed variables. Instead, there are a number of data points which are better than the idealised scenario discussed before in Example 5.1, i.e. points which lie below the diagonal in Figure 5.1.

As illustrated by these observations, simply fixing a certain number of variables of ϕ often does not yield the desired result. Moreover, a straightforward enumeration is not suitable because of the following issues:

Complexity

Even if the number of variables to be fixed is given as say m, there still would be 2^m possible combinations left to try out.

Quality

Proving ϕ with all variable fixed except for one certainly will be very fast, but will hardly give more insight than an aborted verification process. Hence, verification engineers are interested in

5. The SMT-LIB library is composed of various benchmarks to challenge reasoning engines — including many problems from the verification of circuits and systems — and, hence, provides a representative source of problems to be considered within the scope of this work.

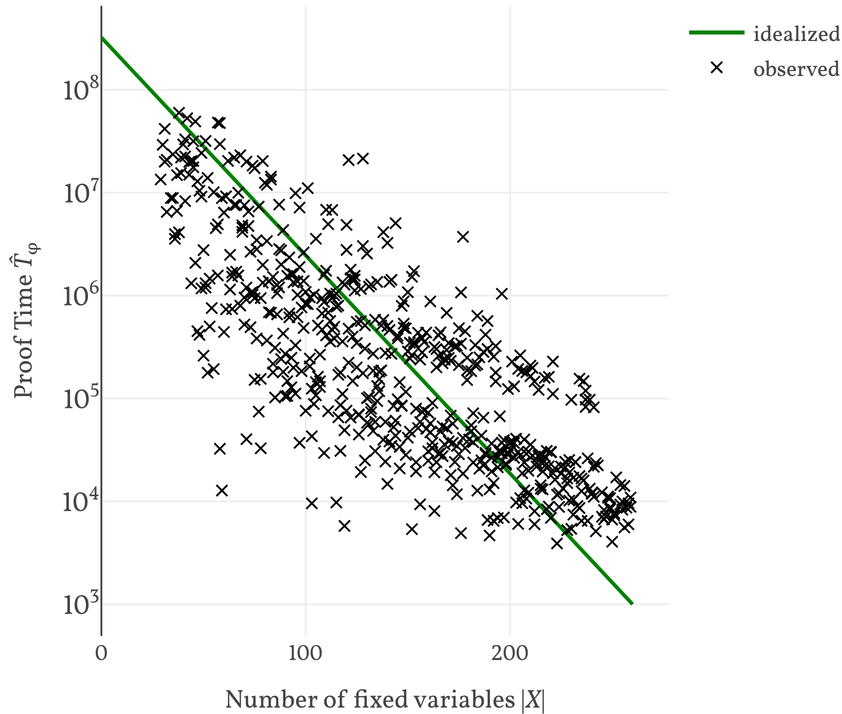


Figure 5.1 Idealised and observed run time of a representative verification problem.

restricting only as little variables as needed before the exponential blow-up kicks in.

Effectiveness

We are particularly interested in verification problems which cannot be completed due to a time-out; here, we are looking for the data points which lie as much to the left in Figure 5.1 as possible but are still below the time-out. These are hard to find by enumeration as one would run into time-outs a lot.

In summary, we are interested in finding the data points in the lower left corner of Figure 5.1, which represent instances where only a small number of variables are fixed (i.e. the instance is as little restricted as possible), while at the same time run time is kept small.

Verification Run Time Analysis 5.2

The observations and discussions from above motivate an analysis of the verification run time in order to determine the best possible data points. This poses an optimization problem which has not been clearly defined so far. In the following, a corresponding definition is provided which is used as a basis for the remainder of this work.

The inputs of the optimization problem are as follows:

- A reasoning engine (such as a SAT solver, SMT solver, or similar) which, given a proposition, either returns true, false or does not terminate.
- A proposition ϕ which takes the time T_ϕ to prove using the reasoning engine.

Note that the actual time units are irrelevant, but we assume that the time is deterministic and reproducible.[6] Furthermore, the proof procedure may not terminate; in that case, T_ϕ is a time unit which is larger than any finite one.

Let $FV(\phi)$ denote the set of free variables occurring in ϕ. Given a subset $X \subseteq FV(\phi)$ of the free variables of ϕ, we define the *average verification run time* $\hat{T}_\phi(X)$ as the average time it takes to prove ϕ with the variables in X set to ground terms, and the rest in $FV(\phi) \setminus X$ kept free. That is, $\hat{T}_\phi(X)$ is the *expected* verification run time if the variables in X are set to an arbitrary fixed value. We have found that, for a given X, we can approximate $\hat{T}_\phi(X)$ with a small number (five) of representative samples.

6. In the experiments summarised below, we use the number of elementary operations of the SMT solver *Z3* [17] as time unit (`rlimit` count), since this is deterministic and independent of architecture or memory.

Example 5.3

Figure 5.1, which has already been discussed before, also provides an illustration of this notation. The figure plots the average verification run time $\hat{T}_\phi(X)$ at the y-axis over the number of fixed variables (i.e. the cardinality $|X|$ of X) at the x-axis for the representative benchmark discussed in Example 5.2. Each data point in the diagram corresponds to $(|X|, \hat{T}_\phi(X))$ for a particular set X of variables to be fixed.

The aim of the analysis is to determine a set X which is as small as possible while still corresponding to a reasonable average verification time. To this end, we need to investigate how the function mapping X to $\hat{T}_\phi(X)$ behaves. With $\emptyset \subseteq X \subseteq FV(\phi)$, we can state that

- $\hat{T}_\phi(FV(\phi))$ is the minimum, because it proves a ground term (no free variables), and
- $\hat{T}_\phi(\emptyset) = T_\phi$ is the maximum, because we prove the original proposition ϕ.

However, in between, the behaviour is not so well defined. From the above, we might guess that the smaller the set X, the larger the average verification run time (i.e. $\hat{T}_\phi(X)$ is anti-monotone over the size of the variable set), but this turns out not be true (also confirmed by Figure 5.1 discussed in Example 5.2). Mathematically, given two different subsets $X, Y \in FV(\phi)$, we have

$$|X| \leq |Y| \not\Longrightarrow \hat{T}_\phi(X) \geq \hat{T}_\phi(Y). \tag{5.1}$$

In other words, increasing the number of fixed variables does not necessarily decrease the average verification run time. This is because variables may depend on other variables, i.e. if we set one of them to a fixed value, the other one is restrained as well.

Hence, the problem remains how to determine an *optimal subset* X of variables (optimal in the sense of smallest set X such that the average verification time $\hat{T}_\phi(X)$ is still acceptable).

Proposed Solution 5.3

The problem motivated and introduced above can be addressed in a number of different ways. A straightforward approach might employ an iterative scheme as follows: Determine the variable with the smallest verification run time (i.e. determine the variable x such that $\hat{T}_\phi(\{x\})$ is minimal). Then, leave it free and determine the variable with the smallest verification run time among the remaining variables. Repeat this process until you reach an average run time which is not feasible anymore. However, because of Equation 5.1, such straight-forward approaches do not lead to satisfactory results in most cases. The optimal solution for say two variables is not necessarily a subset of the optimal solution for three variables (they are not even guaranteed to intersect at all). So, we lack an order structure on the space of possible solutions — all subsets of $FV(\phi)$ — which can guide a search process to the optimal solution. To determine a solution in a rather unstructured space of solutions (such as this one), a number of probabilistic and heuristic approaches are available (e.g. simulated annealing, evolutionary algorithms, etc.). However, all of these need a dedicated *cost function* which unambiguously describes the quality in a quantifiable fashion (i.e. as a number) to guide the search.

To get this cost function, we propose a geometric interpretation of the data points in Figure 5.1. We are looking for the one which is closest to the bottom left corner, i.e. which has the least distance to the origin. Geometrically, if we consider our data points as vectors, we are looking for the vector with the smallest length. In order to make the cost function behave uniformly for different propositions ϕ, we scale both axes with the maximum, i.e. the size of the set of fixed variable

$|X|$ with the total number of free variables $|FV(\phi)|$ and the average verification run time $\hat{T}_\phi(X)$ with the proof time of the original proposition T_ϕ. Thus, for a set X of variables to be fixed, our cost function is

$$q(X) \triangleq \sqrt{\left(\frac{|X|}{|FV(\phi)|}\right)^2 + \left(\frac{\log(\hat{T}_\phi(X))}{\log(T_\phi)}\right)^2}. \tag{5.2}$$

Example 5.4

Figure 5.2 visualises the contours of the cost function (q from Equation 5.2). The theoretical optimum lies at $q(0,0) = 0$. When applied to the results of Figure 5.1, a ranking of the data points becomes apparent, ordering the data points by the distance to the origin (highlighted by solid lines in Figure 5.2). Considering this as cost metric, the optimal solution is the point marked with a green circle in Figure 5.2.

Our cost function requires a concrete value for T_ϕ which can only be approximated (see Section 5.4), as we are considering propositions where T_ϕ is very large (in practice, a time-out). Hence, we need an upper limit for the solutions to consider during the analysis, otherwise we would constantly run into time-outs. Given an upper limit T_{max} which is considered acceptable for the analysis, the *threshold* $\tau(\phi)$ is defined as the number of variables to be fixed such that the average verification run time is still below T_{max}. The value of $\tau(\phi)$ can be efficiently approximated e.g. through a binary search. This confines the number of data points to be considered to the ones which can be analyzed within acceptable run time.

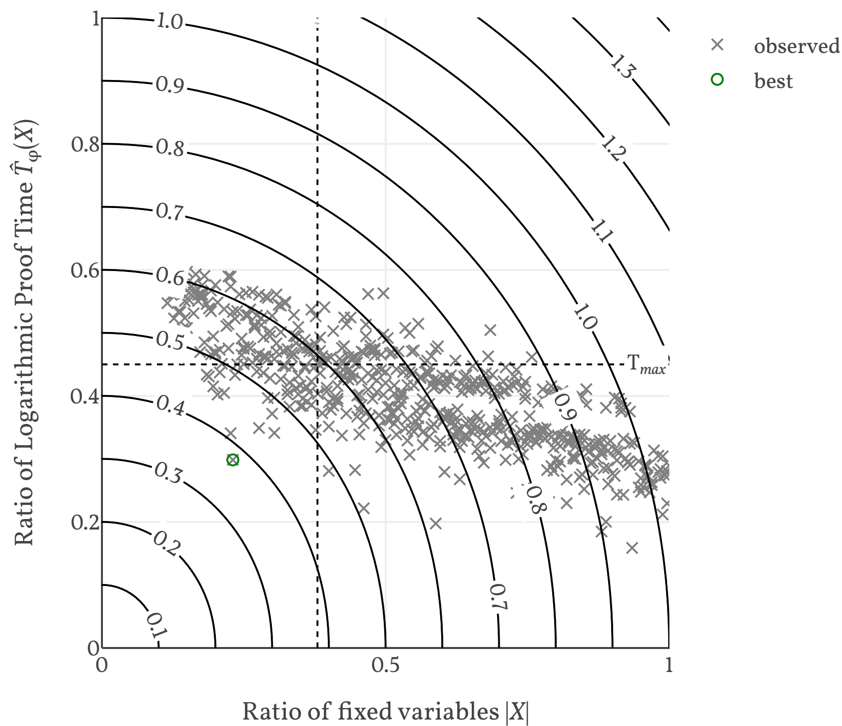


Figure 5.2 Contour of the cost function

Example 5.5

For the example considered above, assume an acceptable time limit T_{max}. Based on this, approximate $\tau(\phi)$ as illustrated by the dotted lines in Figure 5.2. Now, only the data points between the left bottom corner and these lines are considered during analysis. This way, it is ensured that a good solution is derived while, at the same time, the analysis time remains efficient.

Using this cost function and the threshold, any heuristic method of choice can be applied to determine a set X such that $q(X)$ is minimised – this will be our desired solution.

5.4 Implementation

In this section, we describe one possible implementation of the proposed solution described above. As a heuristic, we decided to use *Evolutionary Algorithms* (EAs, [69], [70]) which represent an established method to solve optimization problems, with applications in hardware design [71], [72] or multi-objective optimization [73], [74]. In the following, we briefly review the basic concepts of EAs in general, before discussing how those concepts are utilised in order to address the problem.[7]

5.4.1 Evolutionary Algorithms

Evolutionary algorithms are stochastic search methods inspired by the natural evolution process. The goal is to find a group of individuals (representing solutions) which have the best fitness according to a requested property (in our case, which best satisfy the cost function stated in Equation 5.2).

In order to use EAs for an optimization problem, the following aspects need to be formulated with respect to the considered problem:

Individuals

An individual represents a possible solution for a considered problem, and a set of individuals constitute a population representing a set of solutions. The idea of EAs is that these populations (and hence solutions) are improved over generations.

Mutation operation

Each individual of a population is subjected to mutations which change the solution each individual represents, and hence explore new parts of the search space.

7. However, note that any other optimization methodology can be applied as well, and that the usage of EAs only constitutes a representative.

Recombination operation

Recombinations combine the characteristics of more than one individual, hoping to merge beneficial traits out of them towards a better solution. Recombinations aim for employing changes to individuals in order to explore new parts of the search space as well, but also converge on existing individuals.

Fitness function

After each mutation and recombination, new individuals (the offspring population) are generated. To decide which individuals shall be considered further, a fitness function selects the best individuals and promising candidates for the next generation.

Overall, the typical flow of EAs starts with the generation of an initial population. Afterwards, a sequence of mutation and recombination operations is conducted which yield new generations of populations. The fitness function selects the individuals for the next generation. This process is continued until the process converges (i.e. no real improvements are observed any more) or until a time limit terminates the process.

EA-based Verification Run Time Analysis 5.4.2

In the following, we describe how EAs can be utilised for the optimization problem defined above. Recall that we are interested in keeping the set X of variables to be fixed as small as possible while the average verification run time $\hat{T}_\phi(X)$ remains feasible for the reasoning engine. With this as a basis, we can formulate the different EA aspects with respect to the considered problem as follows:

Individuals A

An individual represents a potential solution X as a bit vector $I = \langle I_i \rangle_{i=1,\ldots,|FV(\phi)|}$ of size $|FV(\phi)|$ such that for every variable

$x_i \in FV(\phi)$ there is a corresponding bit I_i in I which indicates whether $x_i \in X$.

B Mutation

Based on the description of an individual, mutations are performed as follows: Given an individual I and a mutation rate p_m, every bit of its vector is flipped with probability p_M. This leads to a new individual J, representing the new solution.

$$m(I, p_m) = \langle J_i \rangle_{i=1,...,|FV(\phi)|}$$
$$J_i = \begin{cases} \text{flip } I_i & \text{with } p_m \\ I_i & \text{otherwise} \end{cases}$$

C Recombination

Recombinations are performed as follows: Given two individuals I, J and a recombination bias p_c, we combine the two bit vectors by retaining bits which have equal values in both vectors, but randomly choose bits from I or J at positions where they differ. The recombination bias is applied here to prefer one of the individuals. The recombination leads to a new offspring K.

$$r(I, J, p_r) = \langle K_i \rangle_{i=1,...,|FV(\phi)|}$$
$$K_i = \begin{cases} I_i, & \text{if } I_i = J_i \\ I_i, & \text{with } p_c \\ J_i, & \text{otherwise} \end{cases}$$

D Fitness Function

We employ q from Equation 5.2 as the fitness function, with T_ϕ approximated as $T_{max} \cdot 2^{\tau(\phi)}$. $\hat{T}_\phi(X)$ is approximated by averaging the results of a small number of concrete measured times.

E Implementation Aspects

The implementation is available on GitHub:

DFKI-CPS/verification-runtime-analysis

Sources of the DATE 2020 Paper "Verification Runtime Analysis"

The *initial population* is obtained by first approximating the threshold $\tau(\phi)$ with a binary search and then instantiating random individuals with $\tau(\phi)$ positive bits. We employ the algorithm with a very low *mutation rate* p_m, since this yields better recombination results. Individuals to *recombine* are randomly chosen using a normal distribution which prefers the best individuals. In addition, we apply a recombination bias p_r towards the individual with the better score. We monitor the progress of the optimization and spawn increasingly many independent individuals as the optimization slows down. In the beginning, these random individuals are of cardinality $|X|$ where X is the best solution found so far. With every generation which does not yield an improvement over the last, we increase the deviation from $|X|$ in order to avoid getting stuck in a local optimum.

Even though the strategies described here constitute only one possible implementation, it yields very promising results, as the experimental evaluations summarised in the next section will show.

Experiments and Results 5.5

The solution proposed above has been implemented and evaluated using a large corpus of verification instances. This section summarises the most important results obtained by this evaluation. To this end, we first briefly provide details on the actual set-up as well as the considered benchmarks. Afterwards, the obtained results are presented and discussed.

5.5.1 Set-up

As input for the considered problem, we used verification benchmarks provided by the SMT-LIB benchmark library [68] (in the bit vector logic QF_BV), and the SMT solver Z3 [17] as the reasoning engine. The EA has been implemented in the programming language Scala. This language runs on the Java Virtual Machine (JVM), so we use the Java bindings of Z3.

In order to have a deterministic and reproducible notation of time for the analysis, we used Z3's `rlimit` count as a time unit, which provides the number of elementary operations required to solve an instance. This way, the time measurements (required for the fitness function of the EA) remain independent from the actual platform and hardware. The target time-out T_{max} was set to an `rlimit` count of 500 000 which is roughly equivalent to $0.5s$ of computation on the utilised compute server[8].

Using this set-up, the verification run time analysis determines the desired set X out of which the variables to be set to fixed values can be obtained. Afterwards, the originally given proposition ϕ as well as the proposition with the variables in X set to ground terms is solved by Z3 again — showing the impact of the obtained analysis results. For the evaluations, solving times have been measured on an Intel Xeon (E3-1270 v3) compute server with 8 cores and 16 GB of memory running Linux.

5.5.2 Considered Benchmarks

Our methodology is meant for hard verification tasks which do not terminate before a given time-out. The SMT-LIB library provides a huge, representative corpus of such problems from the verification of circuits and systems, constituting a representative source of prob-

8. Note that there is no exact relation between Z3's `rlimit` count and real time since `rlimit` also considers memory operations.

lems for our evaluations. We considered non-iterative quantifier-free bit vector logic (QF_BV) benchmarks from the category "industrial" which are marked as "unsat", where $\tau(\phi)$ (determined by binary search as described above) is larger than 10; the latter ensures that trivial benchmarks which complete in less than roughly $T_{max} \cdot 2^{10} \approx 512s$ are omitted.

With the remaining set of hard benchmarks, the proposed method has been evaluated on a total of 333 propositions. The mean run time t_{A} of the analysis was 86 seconds. 34% (114) of the benchmarks were analyzed in under 60 seconds, 93% (309) finished in less than 10 minutes, and the longest took 23 minutes 37 seconds. There was no significant relation between the run time of the analysis and the original proof time.

Obtained Results 5.5.3

A representative subset of results is summarised in [#tab:rta-results] (For the full results see the linked github repository). Here, the first columns denote the problem size: the number of SMT variables, and the number of bits ($|FV(\phi)|$) representing those SMT variables. The next group of columns shows the results of the analysis: $\tau(\phi)$ is the initially approximated number of variables that has to be fixed, $|X|$ is the size (in bits) of the found solution X; and t_{A} is the run time of the analysis itself. The last column group shows the reduction in verification run time: $T(\phi)$ is the run time with state-of-the-art verification (which results in a time-out for all problems because we explicitly consider hard ones). $\hat{T}_{\phi}^{\mathrm{rnd}}(|X|)$ denotes the run time when just an arbitrary selection of variables $Y \subset FV(\phi)$ with the same size $|Y| = |X|$ is set to a fixed value, while $\hat{T}_{\phi}(X)$ denotes the run time when exactly the variables in X are set to a fixed value.

The results clearly confirm the benefits of our approach. While it is in general not surprising that fixing a number of variables reduces the verification run time, our analysis yields a small number $|X|$ of

variables to fix for maximum effect. By this, verification engineers get much more out of partial verification since it allows them to only set a small portion of the variables to a fixed value. E.g. for *calypto/problem_22.smt2*, a naive method would have led them to set $\tau(\phi) = 128$ variables to a fixed value; with the sophisticated analysis method proposed in this work, just fixing $|X| = 13$ is sufficient — yielding substantially larger coverage.

Moreover, the results confirm that not only the number $|X|$ of variables is important (*how many?*), but also which variables should be set to a fixed value (*which?*). This can clearly be seen in the last two columns of Table 5.1 randomly fixing $|X|$ variables often leads to a time-out ($600s$). In contrast, fixing exactly those variables X obtained by the proposed analysis allows solving *all* benchmarks in negligible run time.

The identified candidates do indeed reduce run time significantly with respect to randomly constrained instances. Out of 333 instances there were 221 which were sped up by factor 10 or more, 167 were sped up by factor 100 or more and 94 were sped up by factor 1000 and more. For 11 benchmarks the reference time could not be determined due to time-outs of factor >10000.

Those benchmarks which were not significantly sped up can actually be recognised during analysis because they show no clear relation between sets of variables and run time and thus have large fluctuations within the population over generations of the EA. We identified several reasons, why this can happen: when too much information is represented by a single SMT variable; when there are only pseudo-random dependencies between variables and when too much of the heavy lifting happens in local function definitions. However, these instances can be quickly identified and might be fixable by adapting the representation of the proof.

Table 5.1 Obtained Results

	Problem Size		Analysis			Verification Run time Reduction		
Benchmark	SMT Variables	$\|FV(\phi)\|$	$\tau(\phi)$	$\|X\|$	t_{A}	$T(\phi)$	$\hat{T}_{\phi}^{\mathrm{rnd}}(\|X\|)$	$\hat{T}_{\phi}(X)$
calypto/problem_22.smt2	33	205	128	13	173s	> 3600*s	> 600.00*s	0.02 s
float/newton.1.3.i.smt2	427	8498	135	33	92s	> 3600*s	> 600.00*s	0.12 s
float/test_v5_r10_vr10_c1_s7608.smt2	855	17860	91	35	298s	> 3600*s	64.15 s	0.16 s
float/test_v5_r15_vr5_c1_s23844.smt2	1280	26710	235	48	492s	> 3600*s	301.85 s	0.23 s
float/test_v7_r12_vr1_c1_s10576.smt2	1431	29853	234	50	525s	> 3600*s	113.98 s	0.26 s
float/test_v7_r17_vr5_c1_s25451.smt2	2024	42194	157	65	326s	> 3600*s	94.39 s	0.36 s
mcm/23.smt2	33	363	10	11	45s	> 3600*s	62.04 s	0.03 s
mcm/63.smt2	36	432	29	12	49s	> 3600*s	155.09 s	0.04 s
mcm/69.smt2	33	396	12	12	47s	> 3600*s	108.72 s	0.04 s
tacas07/Y86_std.smt2	246	5795	700	109	437s	> 3600*s	> 600.00*s	0.07 s
uum/uum16.smt2	190	3428	29	16	27s	> 3600*s	> 600.00*s	0.01 s
uum/uum20.smt2	234	5244	36	20	29s	> 3600*s	> 600.00*s	0.02 s

* = time-out

5.5.4 Further Discussion

The obtained results show how many and which variables to fix to get as much as possible out of partial verification. In this regard, note that there may be external reasons to fix (or not fix) a variable. For example, it makes no sense to fix sensor input which changes rapidly, but it makes a lot of sense to fix configuration parameters which rarely change. Obviously, such considerations can easily be integrated into the proposed analysis e.g. by adding a *weight* to the variables such that instantiating some variables (which do not change often) is favourable to instantiating others (which do change often). In addition to further increase the usefulness of the results, the induced state space of variables should be taken into account, not only considering it's possible values but also the variables that it directly affects. This again, can be easily integrated with weights.

With regard to related work, we need to take a look at the decision heuristics implemented by contemporary SAT- or SMT-solvers, since a lot of effort and research has been invested in their optimisation. However, in a first approach utilising the decision heuristics of MiniSAT and Z3 as a basis, our experiments have shown, that while they are very well suited to identify single variables that have a large impact on the verification time, they fail to identify sets of variables: The identified optimal subset X as identified by our approach doesn't necessarily contain the single most impactful variable identified by the solver (see also Equation 5.1). Explicitly stated, for any two sets of variables X and Y, which is are optimal sets of variables of their respective sizes $|X|$ and $|Y|$ where $|X| < |Y|$, we observed examples, where $|X| \cap |Y| = \emptyset$. This property unfortunately prohibits any straight forward exploitation of such decision heuristics and justifies our approach. The accuracy of our approach despite being less sophisticated than the existing heuristics is not surprising, as our approach is not required to make a very fast decision – like the established heuristics – but rather thoroughly analyses the real runtimes

of the proof engine, leading to an vastly more expensive but also more accurate result. In addition our approach allows to extract the actual average runtimes, giving an indication on the feasibility of the proof on the target system.

The term “partial verification” is also used with model checking, in particular software model checking (see e.g. [76]), referring to techniques to reduce the search space in order to find counterexamples (and, hence, bugs), or referring to the exchange of results between different automatic tools (model checkers, static analyzers, theorem provers) such that the combination of partial results makes the whole verification succeed (see e.g. [78]). This is also referred to as conditional model checking [79]. Furthermore, the term is also used in the context of agents [81], but refers to verification of truthfulness. However, the methodology proposed in this chapter here is not related to any of these previous works and, hence, is novel to the best of our knowledge.

5.6 Conclusion

In this chapter, we presented a systematic verification run time analysis which shows *how many* and *which* variables to fix for maximum verification run time reduction. Experimental evaluations based on a proof-of-concept implementation confirmed the potential and demonstrated that the proposed analysis method does not only yield a partial verification result, but also gets the most out of it. Considering that further analysis methods can be implemented on top of this methodology, this work provides a promising basis for future work in this direction.

In the context of self-verification this analysis method is able to indicate to the designer, which parts of a proof are worthy to transfer or postpone into run-time and by this enhances the workflow.

6 The Future of Self-Verification

Self-Verification is an entirely new approach to the verification of cyber-physical and embedded systems and opens up its own field of potential research directions. We present four promising ideas in this chapter.

6.1 Predictive Self-Verification

If neither safety nor functionality can be sacrificed, whenever a trigger transition is reached during the run-time of a system, the system must pause until it is proven safe to operate again (See also Section 3.3.2). This may lead to unacceptable delays in the execution. One possible remedy could be to utilise the idle-time of the proof system to proactively prove possible future trigger transitions. However, this imposes a couple of hard problems:

(1) We need a method to predict future transitions and
(2) we need to cache discharged proofs.

The prediction of future transitions in cyber-physical systems is completely dependent on the scope of the system and may involve arbitrary scientific disciplines, as such systems interact with the physical world. Assume a self-driving vehicle as an example: We may use

meteorologic information to predict the static and adhesive friction of the road due to rain, ice and wind conditions. We might use statistical traffic data for the prediction of traffic jams, psychological approaches to intention recognition to predict the behaviour of the other road users, etc. It becomes clear that this direction is by no means a venture that can be easily taken. Not because these methods do not exist, but because complex techniques from numerous scientific disciplines have to be combined. However, even rudimentary approaches can give us a benefit. We utilise "free" idle time between trigger transitions and every *hit* will improve the responsivenes of the sytem.

Even with a suitable method of prediction, problem (2), caching of discharged proofs, remains. We are working with huge state spaces (recall, that they were too large to handle prior to deployment) and over time, an arbitrarily large amount of discharged proofs may accumulate (by this, too large to keep in memory). There are two ideas to cope with this (which may be combined):

(1) We may employ "garbage collection" which clears proofs for states that were predicted under entirely different conditions but stayed unused.
(2) Proof results could be condensed by proving patterns of states and storing only the pattern.

Overall, predictive verification is a promising approach to increase the availability and responsiveness of self-verifying systems.

Just-in-Time Verification 6.2

So far, we have worked with statically assigned trigger transitions such as a change of the configuration or an environment variable. If these variables are under the control of the system this allows for a completely safe system while we might sacrifice some functionality (when a proof fails) or have to accept delays (when a proof consumes more time than predicted). However, variables that are externally

controlled might leave the system unverified for a period of time or even proven erroneous during execution. For instance, a self-driving car may encounter unexpected road conditions, traffic signals, or pedestrians that require immediate verification of its safety and functionality. In such cases, waiting for a correctness proof to complete before resuing the execution may not be feasible or desirable. In addition, verifiying properties of the system over general unbounded time intervals may be undecidable or intractable, as it often involves checking properties over infinite non-conflatable state spaces.

As a remedy for these shortcomings, we propose a novel approach called *just-in-time verification*, which dynamically triggers *time-bounded verifications* during run-time of the system. Time-bounded verification is a method of checking the correctness of real-time systems over time intervals of fixed, bounded length. It is useful for verifying properties that are relevant only for a certain duration, such as deadlines, timeouts, or response times. However, in our use-case time-bounded verification can also be a means to avoid some of the undecidability and complexity issues of unbounded verification [82]. Time-bounded verification can be applied to various logics and models of real time, such as timed automata, metric temporal logic, event-clock automata, and perturbed timed automata. It can also be combined with other techniques, such as partial-order reduction, preemption locks, and priority locks, to improve the efficiency and scalability of the verification process. [83]

The idea of just-in-time verification is to monitor the system state and the environment variables continuously, and to repeatedly conduct time-bounded verifications. The verification is performed over a time interval that is sufficient to ensure the correctness of the system for a time span long enought to finish the proof of the next time period. If the verification succeeds, the system can proceed with the execution. If the verification fails, the system can fall-back to a safety measure.

Prerequisites 6.2.1

Just-in-time verification imposes two essential requirements that have to be met during the design of the system:

The central requirement is the existence of a *safety measure* that can be executed in any state of the system and bring the system into a stable safe state. E.g. for a vehicle this might be pulling over and coming to a safe stop or requesting human intervention. The time t_{safe} it takes to execute this safety measure may be dynamically dependent on the state of the system (e.g. the speed of a vehicle) but it (or its upper bound) must be predictable from any given state.

In addition, we need a heuristic $\eta(t, \sigma)$ that gives us a conservative estimate of the timespan we can prove from the current state σ of the system into the future, given the available proof time t. The results of Chapter 5 indicate that such a heuristic could be inferrable. The recent dramatic advances in machine learning, especially deep neural networks, offer another encouragement to investigate this. Even a very basic heuristic will never leave the system in an unsafe state, but might cause inconvenience, as the system will trigger unnecessary emergency stops (i.e. the safety measure).

Operation 6.2.2

For its operation a just-in-time verified system has to be equipped with a self-verification system, composed of a monitor, a verifier and a controller. The self-verification system is responsible for verifying the system dynamically during operation, as well as for adapting the system behavior accordingly.

The *monitor* component observes the state of the system and the environment. The monitor can be implemented e.g. with sensors, event logs or access to internal state of the deployed system. The monitor communicates the information to the *verifier* component, which runs the verification tasks of the system using time-bounded verification.

The verifier can e.g. utilise an SMT-solver or model checker. The verification results are communicated to the *controller*, which has the power to trigger the safety measure at any time of the system execution, as well as resume normal execution of the system.

After the system is deployed or when the safety measure has been triggered, the proof system has to be bootstrapped by proving a timespan t_0 which ends at $\dashv_{t_0}$. When this is successful, the controller can can start or resume the normal operation of the system. During operation, the proof engine will continuously trigger proofs in a loop follwing the schema:

(1) the proof for time span t_{n-1} completes.
(2) we trigger a proof for time span t_n with $\dashv_{t_n} > \dashv_{t_n - 1} + t_{\text{safe}}$. The proof itself consumes the timespan t_n^{proof}. t_n is determined by our heuristic η.
(3) when the proof completes before $\dashv_{t_n} - t_{\text{safe}}$ we can continue from (1) (Figure 6.1). This is the the stable just-in-time proof cycle.
(4) when verifier proves the violation of a property or does not yield a proof result in time (i.e. before $\dashv_{t_n} - t_{\text{safe}}$) the controller interrupts the proof cycle by executing the safety measure. (Figure 6.2, Figure 6.3)
(5) when the system reaches a safe state after execution of the safety measure, the verifier can bootstrap the proof cycle again and start from (1)

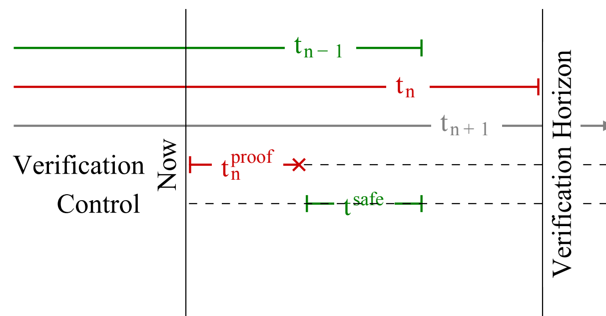


Figure 6.2 Reasons to exit the proof cycle: an error is detected before $t_{n-1} - |t_{\text{safe}}|$. The system starts the emergency measure just in time

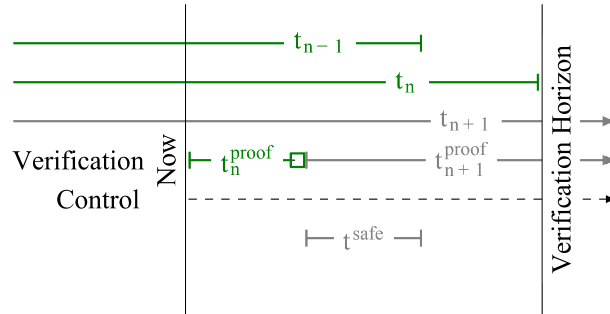


Figure 6.1 The ideal jit scenario: . t_{n-1} is the timespan that was previously verified. In this ideal scenario t_n^{proof} ends before the end of $\dashv_{t_{n-1}} - |t_{\text{safe}}|$. Operation can continue and the next proof $(n+1)$ is triggered.

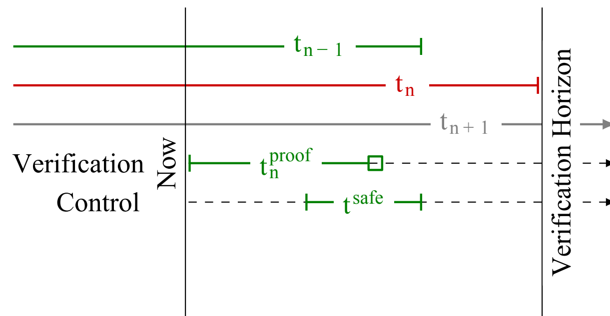


Figure 6.3 Reasons to exit the proof cycle: the proof does not complete before $t_{n-1} - |t_{\text{safe}}|$, hence the result will not be usable even if proven correct. The system starts the emergency measures just in time

The proof cycle is always safe, but can in the worst case restrict functionality of the system completely if the heuristic η always overestimates the time t that can be proven or the proof is never faster than the execution. Hence, the unanswered research questions that arises from this approach are:

(1) Is there a reliable method to infer a "good enough" η for a given system?
(2) Are there real-world systems to which just-in-time verification can be applied without severely restricting functionality?

We have implemented experiments, namely a simplified simulation of an adaptive cruise control and several planning algorithms but until now failed to find the sweet spot, where a stable proof cycle can be

permanently established while also verifying interesting properties and operating in real time. However, we still believe, that it is worth investigating further as just-in-time verification has several advantages over the static trigger-based approach:

(1) It can ensure safety even in the presence of uncertain and unpredictable environments.
(2) It can drastically reduce the verification overhead and the execution delay by focusing on the relevant scenarios and properties.
(3) It can increase the availability and responsiveness of the system by avoiding unnecessary delays due to long running verification tasks.
(4) It could enable the evolution and integration of the system by allowing the verification of new components.

6.3 Dependent Operation

In the context of just-in-time verification, we propose *Dependent Operation* as a critical concept, that may aid in establishing a stable proof cycle. Particularly for systems where operational parameters directly influence the verification process or the safety measure. This is exemplified by an autonomous vehicle:

- The speed of an autonomous vehicle is proportional to the time it takes to bring the vehicle to a safe stop (i.e. t_{safe} reduces).
- The speed of an autonomous vehicle also exponentially reduces the search space of time-bounded veficiation tasks as the surface of potential positions of the vehicle within the relevant time interval shrinks. (i.e. verification time reduces)

This observation could be the key to practically applicable just-in-time verification: The system adapts its operation to the proof system and vice versa. By this, both system availability and safety can be maximised dynamically.

A continuous feedback loop is integral to this process. It involves real-time monitoring of both the system's state and the environment. Adjustments to operational parameters are based on this ongoing assessment. However, implementing Dependent Operation poses several challenges:

(1) Real-time Data Analysis: The system must efficiently process and analyze data in real-time to make informed adjustments. This requires advanced algorithms capable of quick and accurate decision-making.
(2) Predictive Accuracy: The system's ability to predict the implications of parameter adjustments on verification outcomes is crucial. Ensuring the accuracy of these predictions is paramount, particularly in safety-critical scenarios.
(3) Balancing Safety and Operational Efficiency: It is vital to strike a balance between maintaining operational efficiency and ensuring safety. Overly conservative adjustments might hinder system performance, while aggressive adjustments could compromise safety.

The application of dependent operation holds significant promise in enhancing both the safety and efficiency of complex, real-time systems. Its adaptability makes it suitable for a wide range of applications beyond autonomous vehicles, including industrial automation and smart infrastructure management.

6.4 Verification Aware Inference

Inferred systems or components of systems, which are the result of modern machine learning approaches (e.g. deep neural networks) are usually black boxes for verification methods. Their quality assurance usually focuses on the training inputs. There are several first attempts to the verification of such trained models [84], [85]. However, they share the problem of non-scalability. With the recent advent of transformer models, it seams feasible to train models not to produce

continuous single decisions based on the current state but instead repeatedly update the actual plan or code that is suitable to handle the current situation of the system based on the observation of the environment.

If we take into account the observations from Section 6.3, that controllable variables can not only have an effect on t_{safe} but also on the search space, we can imagine an inferred system that is optimised to produce plans that not only serve to perform their primary tasks but also reduce verification time, so that it becomes feasible to establish a proof cycle as outlined in Section 6.2. Transformer models respond very well to feedback (e.g. Reinforcement learning from human feedback) and by this might be optimisable to produce plans that are easy to verify, by using the verification time (and result) as basis for feedback to the transformer model. For complex systems with (for humans) impenetrable inter-dependencies this might well be the only way to establish dependent operation at all.

6.5 Conclusion

In this chapter, we have explored potential future directions of Self-Verification in cyber-physical and embedded systems, highlighting innovative approaches like Predictive Self-Verification, Just-in-Time Verification, Dependent Operation, and Verification Aware Inference.

The presented concepts relate to the principles of any-time algorithms [86], as they offer valid outputs even if interrupted and the quality of the response increases with the available computation time. However, we believe that just-in-time verification may be a simpler concept that allows for a clearer devision of implementation and verification as the verification system can be integrated in a late phase of the development, while any-time algorithms impose increased development costs in the early implementation phase.

The exploration of Self-Verification in this chapter could represent a significant shift in how we approach the verification of complex systems in the future, moving towards more dynamic and responsive methodologies. This is in response to the increasing complexity and autonomy of contemporary systems but also an acknowledgment of the need for adaptive verification strategies. The concepts and methodologies discussed are also indicative of a broader movement towards seamlessly embedding verification into system operation.

7 Conclusion

This thesis, has elaborated on self-verification — systems which are not verified a priori, during the design phase, but where the proof obligations incurred during the development are postponed until after deployment, and are proven at run-time. This makes proofs easier, as we can instantiate a number of the parameters of the system which are unknown at design time but become known at run-time. This reduces the state space, turning the exponential growth of the state space — the bane of model-checking — into exponential reduction. Self-verification is supported by a tool chain we have developed, which allows specification in SysML/OCL, system modelling in Clash, and verification using SMT provers and SAT checkers.

Despite the use of specific modelling languages and prover tools in our evaluation, the basic idea is independent of these. As long as we can translate the verification conditions into a format where we can track the variables to be instantiated into run-time, and which is suitable to automatic proof, the approach is viable and competitive, because of the exponential reduction of the search space.

We have investigated the implications, self-verification has on the development and were able to establish a core set of development rules regarding the applicability of the approach as well as design decisions that should be made:

- Self-verification is especially well suited for systems, that may sacrifice availability for safety. This explicitly excludes fail-safe systems but nevertheless includes a large portion of application domains.
- Proof obligations should be discharged as early as possible but as late as needed. For this a minimal set of *trigger transitions* should be selected, which defines the transitions of the system that initiate a verification task.
- Trigger transitions must not themselves be safety critical: the non-execution or delay of a trigger transition should never violate the specification — if we verify our parachute upon pulling the release cord it is clearly too late to handle failure.
- Trigger transitions should maximise the reduction in prover run time. A trigger transition which does not significantly reduce the verification run time, may as well be left as a normal transition.

For the latter point, we have developed a methodology which can indicate which sets of variables of a proof offer the largest reduction in verification run time, aiding the designer in this decision. We demonstrated the practical application of the methodology through a proof-of-concept implementation and extensive experimental evaluation, offering a novel perspective on managing computational resources in post-deployment verification tasks, emphasizing the strategic fixation of variables to balance verification coverage and computational efficiency.

Note that self-verification is not meant to *replace* the existing verification flow but rather *enhances* it. We should by all means use all well-known powerful tools at design time to discharge verification conditions as before. However, self-verification offers a different way to tackle proof obligations which cannot be shown at design time, supplementing design time verification, and offering the designer to pick the best of all possible worlds.

7.1 Contributions

This cumulative thesis offers significant contributions to the field of embedded system verification that span conceptual frameworks, methodological advancements, and empirical validations:

Chapter 3 delved into the fundamentals of self-verification, presenting a novel design and verification methodology that defers a portion of the verification process until after deployment. This methodology leverages the specific operating context of systems to reduce the complexity of verification tasks. Several case studies, based on an implementation that uses a lightweight SAT solver not only validate the proposed methodology but also demonstrate its practical applicability and effectiveness.

Chapter 4 further explores the application of self-verification within the system development lifecycle, specifically through the lens of design time verification, run-time verification, and their relationship with self-verification. By classifying system transitions and optimizing verification efforts, the chapter presents a nuanced model that enhances system safety and flexibility without compromising on functionality. A detailed case study on an access control system exemplifies the real-world applicability of self-verification, providing methodological insights.

Chapter 5 presents a novel approach to determining the optimal number and selection of variables to be fixed for verification tasks, employing evolutionary algorithms. The effectiveness of this methodology is confirmed by experimental evaluations, which demonstrate significant reductions in verification run times over random variable selections. It goes beyond the capabilities of established decision heuristics as it considers the real runtimes of proofs for sets of variables.

Together, this thesis contributes a comprehensive framework for self-verification that encompasses conceptual foundations, practical implementation strategies, and empirical validations. It proposes innovative methodologies and tools for optimizing verification processes, and validates these approaches through extensive evaluations. These contributions mark a significant step forward in the design, verification, and maintenance of ebedded systems, with the potential to have a lasting impact on the field.

Future Work 7.2

The idea of self-verification opens up numerous possiblities for future work, which did not fit the scope of this thesis:

More thorough empirical studies and real-world deployments of self-verification methodologies would provide valuable insights into their practicality and effectiveness. This includes conducting large-scale evaluations across diverse application domains, from critical infrastructure to consumer electronics, to assess the adaptability and resilience of self-verification-enabled systems. Such studies would not only validate theoretical models and simulation results but also identify practical challenges and opportunities for refinement.

To leverage the concept even further, it would be highly beneficial to implement the verification engines as dedicated hardware components, maybe even specifically tailored for the concrete proof instances.

A natural progression from the foundational work laid out in this thesis is the exploration of more sophisticated self-verification frameworks that can dynamically adjust to changing operational contexts and system states (as outlined in Chapter 6). This involves developing adaptive algorithms that can not only verify but also optimise system configurations in real-time, enhancing system reliability and performance without human intervention. Research could

focus on creating models that predict configuration changes and assess their impact on system behavior, ensuring that self-verification processes remain efficient and effective under varying conditions.

Furthermore, the exploration of decentralised self-verification approaches for distributed systems offers significant potential. In environments where systems consist of multiple interacting components, such as in the Internet of Things (IoT) or distributed ledger technologies, traditional centralised verification approaches as illuminated in this thesis may not be feasible. Research could focus on developing distributed self-verification protocols that allow individual components to independently verify their operation while contributing to the overall system's integrity. This would require novel consensus mechanisms and trust models to ensure that verification processes are reliable and tamper-resistant.

7.3 Conclusion

Despite the required future research, the evaluations and discussions in this thesis show that, following the proposed ideas, allows for a novel verification methodology that can be adapted today, which does not rely on incremental improvement of existing tools and methods but tackles complexity from a completely different angle. While self-verification offers immense potential, it also comes with challenges and limitations that must be addressed in the future. Balancing optimism with a realistic evaluation of these challenges, further research as well as collaborative efforts between academia and industry are necessary to find the sweet spots in delaying of verification tasks. This will be crucial in advancing self-verification, pushing the boundaries of systems reliability, safety, and efficiency in an increasingly automated world.

References

[1] M. Ring, J. Stoppe, C. Lüth, and R. Drechsler, "Change impact analysis for hardware designs — from natural language to system level," in *Forum on Specification & Design Languages (FDL 2016)*, Bremen, Germany, Sep. 2016, pp. 1–7, doi: https://doi.org/10.1109/FDL.2016.7880369.

[2] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Better Late Than Never — Verification of Embedded Systems After Deployment," in *Design, Automation Test in Europe Conference Exhibition (DATE 2019)*, Florence, Italy, Mar. 2019, pp. 890–895, doi: https://doi.org/10.23919/DATE.2019.8714967.

[3] M. Ring and C. Lüth, "Let's Prove It Later — Verification at Different Points in Time," in *International Conference on Software Engineering and Formal Methods (SEFM 2019)*, Oslo, Norway, Sep. 2019, pp. 454–468, doi: https://doi.org/10.1007/978-3-030-30446-1_24.

[4] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Verification Runtime Analysis — Get the Most Out of Partial Verification," in *Design, Automation Test in Europe Conference Exhibition (DATE 2020)*, Grenoble, France, Mar. 2020, pp. 873–878, doi: https://doi.org/10.23919/DATE48585.2020.9116543.

[5] J. Yuan, C. Pixley, and A. Aziz, *Constraint-Based Verification*. Springer, 2006.

[6] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC Verification library," in *Forum on Specification & Design Languages (FDL 2009)*, Sophia Antipolis, France, Sep. 2009, pp. 1–6.

[7] E. M. Clarke Jr., O. Grumberg, and D. A. Peled, *Model Checking*, 2nd ed. MIT Press, 2018.

[8] A. Koczor, L. Matoga, P. Penkala, and A. Pawlak, “Verification approach based on emulation technology,” in *International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS 2016)*, Kosice, Slovakia, Apr. 2016, pp. 169–174, doi: https://doi.org/10.1109/DDECS.2016.7482447.

[9] G. E. Moore, “Cramming More Components Onto Integrated Circuits,” *Proc. IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998, doi: https://doi.org/10.1109/jproc.1998.658762.

[10] R. N. Charette, “This car runs on code,” *IEEE spectrum*, vol. 46, pp. 3, Apr. 2009.

[11] R. Drechsler, M. Soeken, and R. Wille, “Formal Specification Level — Towards Verification-Driven Design Based on Natural Language Processing,” in *Forum on Specification & Design Languages (FDL 2012)*, Vienna, Austria, Sep. 2012, pp. 53–58.

[12] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification — A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2007.

[13] N. Eén and N. Sörensson, “An Extensible SAT-solver,” in *International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003, pp. 502–518, doi: https://doi.org/10.1007/978-3-540-24605-3_37.

[14] M. Bozzano *et al.*, “The MathSAT 3 System,” in *International Conference on Automated Deduction (CADE 2005)*, Tallinn, Estonia, Jul. 2005, pp. 315–321, doi: https://doi.org/10.1007/11532231_23.

[15] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, “SWORD: A SAT like prover using word level information,” in *IFIP International Conference on Very Large Scale Integration (VLSI-SoC 2007)*, Atlanta, GA, USA, Oct. 2007, pp. 88–93, doi: https://doi.org/10.1109/VLSISOC.2007.4402478.

[16] B. Dutertre, “Yices 2.2,” in *International Conference on Computer Aided Verification (CAV 2014)*, Vienna, Austria, Jul. 2014, pp. 737–744, doi: https://doi.org/10.1007/978-3-319-08867-9_49.

[17] L. M. de Moura and N. Bjørner, “Z3 — An Efficient SMT Solver,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Budapest, Hungary, Mar. 2008, pp. 337–340, doi: https://doi.org/10.1007/978-3-540-78800-3_24.

[18] H. D. Foster, “Why the design productivity gap never happened,” in *International Conference on Computer-Aided Design (ICCAD 2013)*, San Jose, CA, USA, Nov. 2013, pp. 581–584, doi: https://doi.org/10.5555/2561828.2561943.

[19] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May 2009, doi: https://doi.org/10.1016/j.jlap.2008.08.004.

[20] R. Drechsler, M. Fränzle, and R. Wille, “Envisioning Self-Verification of Electronic Systems,” in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, Jun. 2015, pp. 1–6, doi: https://doi.org/10.1109/ReCoSoC.2015.7238101.

[21] D. L. Perry, *VHDL: Programming by Example*. McGraw-Hill Education, 2002.

[22] D. Thomas and P. R. Moorby, *The Verilog® Hardware Description Language*. Springer New York, 2002.

[23] R. Drechsler, M. Soeken, and R. Wille, “Formal Specification Level,” in *Models, Methods, and Tools for Complex Chip Design*, vol. 265, J. Haase, Ed. Springer, 2014, pp. 37–52.

[24] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel, “Object-oriented modeling and synthesis of SystemC specifications,” in *Asia and South Pacific Design Automation Conference (ASP-DAC 2004)*, Yokohama, Japan, Jan. 2004, pp. 238–243, doi: https://doi.org/10.1109/ASPDAC.2004.1337573.

[25] J. Bachrach *et al.*, “Chisel — constructing hardware in a Scala embedded language,” *Design Automation Conference (DAC 2012)*. ACM Press, pp. 1216–1225, Jun. 2012, doi: https://doi.org/10.1145/2228360.2228584.

[26] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards, “ClaSH: Structural Descriptions of Synchronous Hardware using Haskell,” in *EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD 2010)*, Lille, France, Sep. 2010, pp. 714–721, doi: https://doi.org/10.1109/DSD.2010.21.

[27] L. J. Hafer and A. C. Parker, “A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, no. 1. IEEE, pp. 4–18, Jan. 1983, doi: https://doi.org/10.1109/tcad.1983.1270016.

[28] J.-R. Abrial, “System study: Method and example,” 1999. Accessed: Mar. 17, 2021. [Online]. Available: http://atelierb.eu/ressources/PORTES/Texte/porte.anglais.ps.gz.

[29] “Systems Modeling Language (SysML), Version 1.6,” Object Management Group, Dec. 2019. [Online]. Available: https://www.omg.org/spec/SysML/1.6/PDF.

[30] The Eclipse Foundation, "EMF Diff/Merge," 2012. http://www.eclipse.org/diffmerge/ (accessed Mar. 18, 2021).

[31] L. C. Briand, Y. Labiche, and L. Sullivan, "Impact analysis and change management of UML models," in *International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sep. 2003, pp. 256–265, doi: https://doi.org/10.1109/ICSM.2003.1235428.

[32] M. Jarke, "Requirements tracing," *Communications of the ACM*, vol. 41, no. 12. ACM, pp. 32–36, Dec. 1998, doi: https://doi.org/10.1145/290133.290145.

[33] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: an efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, Kanazawa, Japan, Oct. 2004, pp. 148–154.

[34] FZI Karlsruhe, "KaSCPar - Karlsruhe SystemC Parser Suite," 2012. http://www.fzi.de/sim/kascpar.html (accessed Mar. 07, 2018).

[35] J. Castillo, P. Huerta, and J. I. Martinez, "An open-source tool for SystemC to Verilog automatic translation," *Latin American Applied Research*, vol. 37, no. 1, pp. 53–58, 2007.

[36] C. Brandolese, P. Di Felice, L. Pomante, and D. Scarpazza, "Parsing SystemC — an open-source, easy-to-extend parser," in *IADIS International Conference on Applied Computing (AC 2006)*, San Sebastian, Spain, Feb. 2006, pp. 706–709.

[37] D. Berner, J.-P. Talpin, H. Patel, D. A. Mathaikutty, and S. Shukla, "SystemCXML — An extensible SystemC front end using XML," in *Forum on Specification & Design Languages (FDL 2005)*, Lausanne, Switzerland, Sep. 2005, pp. 405–409.

[38] C. Genz and R. Drechsler, "Overcoming limitations of the SystemC data introspection," in *Design, Automation Test in Europe Conference Exhibition (DATE 2009)*, Nice, France, Apr. 2009, pp. 590–593, doi: https://doi.org/10.1109/DATE.2009.5090734.

[39] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip," in *Conference on Embedded software (EMSOFT 2005)*, Jersey City, NJ, USA, Sep. 2005, pp. 317–324, doi: https://doi.org/10.1145/1086228.1086286.

[40] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient Automatic Visualization of SystemC Designs," in *Forum on Specification & Design Languages (FDL 2003)*, Frankfurt, Germany, Sep. 2003, pp. 646–658.

[41] J. Stoppe, R. Wille, and R. Drechsler, "Data extraction from SystemC designs using debug symbols and the SystemC API," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2013)*, Natal, Brazil, Nov. 2013, pp. 26–31, doi: https://doi.org/10.1109/ISVLSI.2013.6654618.

[42] B. Meyer, "Applying 'design by contract,'" *Computer*, vol. 25, no. 10. IEEE, pp. 40–51, Oct. 1992, doi: https://doi.org/10.1109/2.161279.

[43] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Sangiovanni Vincentelli, "Contract-Based Design for Computation and Verification of a Closed-Loop Hybrid System," in *International Workshop on Hybrid Systems: Computation and Control (HSCC 2008)*, St. Louis, MO, USA, Apr. 2008, pp. 58–71, doi: https://doi.org/10.1007/978-3-540-78929-1_5.

[44] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions — Advanced Specification and Verification with JML and ESC/Java2," in *International Conference on Formal Methods for Components and Objects (FMCO 2005)*, Amsterdam, The Netherlands, 2006, pp. 342–363, doi: https://doi.org/10.1007/11804192_16.

[45] P. Baudin *et al.*, "ACSL: ANSI/ISO C Specification Language Version 1.16," INRIA, 2020. [Online]. Available: http://frama-c.com/download/acsl.pdf.

[46] F. L. Bauer *et al.*, "Towards a wide spectrum language to support program specification and program development," *ACM SIGPLAN Notices*, vol. 13, no. 12, pp. 15–24, Dec. 1978, doi: https://doi.org/10.1145/954587.954588.

[47] J.-R. Abrial and C. A. R. Hoare, *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[48] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International journal on software tools for technology transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[49] B. Dion and J. Gartner, "Efficient Development of Embedded Automotive Software with IEC 61508 Objectives using SCADE Drive," in *VDI 12th International Conference: Electronic Systems for Vehicles*, 2005, pp. 1427–1436.

[50] M. Richters and M. Gogolla, "OCL: Syntax, Semantics, and Tools," in *Object Modeling with the OCL*, T. Clark and J. Warmer, Eds. Springer, 2002, pp. 42–68.

[51] R. Drechsler, S. Autexier, and C. Lüth, "Model-Based Specification and Refinement for Cyber-Physical Systems," in *Dynamics in Logistics*, Springer International Publishing, 2016, pp. 3–17.

[52] "Standard SystemC Language Reference Manual," in *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, IEEE, 2012.

[53] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[54] A. Lanusse *et al.*, "Papyrus UML — an open source toolset for MDA," in *European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, Enschede, the Netherlands, Jun. 2009, pp. 1–4.

[55] S. Autexier and N. Müller, "Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents," in *ACM Symposium on Document Engineering (DocEng2010)*, Manchester, UK, Sep. 2010, pp. 97–106, doi: https://doi.org/10.1145/1860559.1860580.

[56] S. Autexier, "Similarity-Based Diff, Three-Way-Diff and Merge," *International Journal of Software and Informatics*, vol. 9, no. 2, pp. 259–277, Aug. 2015.

[57] "Object Constraint Language (OCL), Version 2.4," Object Management Group, Feb. 2014. [Online]. Available: https://www.omg.org/spec/OCL/2.4/PDF.

[58] M. Soeken *et al.*, "Quality Assessment for Requirements based on Natural Language Processing," in *Forum on Specification & Design Languages (FDL 2014)*, Munich, Germany, 2014, pp. 1–4.

[59] F. Bornebusch, R. Wille, and R. Drechsler, "Towards Lightweight Satisfiability Solvers for Self-Verification," in *International Symposium on Embedded Computing and System Design (ISED 2017)*, Durgapur, Dec. 2017, pp. 1–5, doi: https://doi.org/10.1109/ISED.2017.8303924.

[60] A. Balint and U. Schöning, "Engineering a Lightweight and Efficient Local Search SAT Solver," in *Algorithm Engineering — Selected Results and Surveys*, Springer, 2016, pp. 1–18.

[61] T. Ivan and E. M. Aboulhamid, "An Efficient Hardware Implementation of a SAT Problem Solver on FPGA," in *EUROMICRO Conference on Digital System Design (DSD 2013)*, Los Alamitos, CA, USA, Sep. 2013, pp. 209–216, doi: https://doi.org/10.1109/DSD.2013.31.

[62] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler, "SAT-Lancer: A Hardware SAT-Solver for Self-Verification," in *ACM Great Lakes Symposium on VLSI (GLVLSI 2018)*, Chicago, IL, USA, May 2018, pp. 479–482, doi: https://doi.org/10.1145/3194554.3194643.

[63] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May 2009.

[64] A. Hejlsberg, “TypeScript Version 4.2,” 2021. https://www.typescriptlang.org (accessed Apr. 11, 2021).

[65] D. Essame, J. Arlat, and D. Powell, “Available fail-safe systems,” Tunis, Tunisia, Oct. 1997, doi: https://doi.org/10.1109/ftdcs.1997.644721.

[66] A. Biere, “PicoSAT Essentials,” *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, pp. 75–97, 2008.

[67] R. Brummayer and A. Biere, “Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, York, UK, 2009, pp. 174–177, doi: https://doi.org/10.1007/978-3-642-00768-2_16.

[68] “Official SMT-LIB repository,” 2018. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks (accessed Dec. 14, 2020).

[69] D. B. Fogel, “Evolutionary algorithms in theory and practice,” *Complexity*, vol. 2, no. 4, pp. 26–27, 1997.

[70] Z. Michalewicz and M. Schoenauer, “Evolutionary Algorithms for Constrained Parameter Optimization Problems,” *Evolutionary Computation*, vol. 4, no. 1, 1996.

[71] B. Korousic-Seljak, J. Silc, and G. Papa, “An Evolutionary Approach to Problems in Electrical Engineering Design,” 2005.

[72] Z. Vasícek and L. Sekanina, “Evolutionary Approach to Approximate Digital Circuits Design,” *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 432–444, Jul. 2015, doi: https://doi.org/10.1109/TEVC.2014.2336175.

[73] R. Chen, K. Li, and X. Yao, “Dynamic Multiobjectives Optimization With a Changing Number of Objectives,” *IEEE Transactions on Evolutionary Computation*, vol. 22, pp. 157–171, Feb. 2018, doi: https://doi.org/10.1109/TEVC.2017.2669638.

[74] K. Deb, “Multi-Objective Evolutionary Algorithms,” in *Handbook of Computational Intelligence*, J. Kacprzyk and W. Pedrycz, Eds. Springer, 2015.

[75] P. Parizek and F. Plasil, “Partial Verification of Software Components: Heuristics for Environment Construction,” in *EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, Lübeck, Germany, Aug. 2007, pp. 75–82, doi: https://doi.org/10.1109/EUROMICRO.2007.46.

[76] A. Groce and W. Visser, “Heuristics for model checking Java programs,” *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 4, pp. 260–276, Aug. 2004, doi: https://doi.org/10.1007/s10009-003-0130-9.

[77] V. Wüstholz, “Partial Verification Results,” PhD dissertation, ETH Zürich, 2015.

[78] D. Beyer, “Partial Verification and Intermediate Results as a Solution to Combine Automatic and Interactive Verification Techniques,” in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques (ISoLA 2016)*, Corfu, Greece, Oct. 2016, pp. 874–880, doi: https://doi.org/0.1007/978-3-319-47166-2_60.

[79] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, “Conditional Model Checking — A Technique to Pass Information Between Verifiers,” in *International Symposium on the Foundations of Software Engineering (FSE 2012)*, Cary, NC, USA, Nov. 2012, pp. 1–11, doi: https://doi.org/10.1145/2393596.2393664.

[80] I. Caragiannis, E. Elkind, M. Szegedy, and L. Yu, “Mechanism Design: From Partial to Probabilistic Verification,” in *ACM Conference on Electronic Commerce (EC 2012)*, New York, NY, USA, Jun. 2012, pp. 266–283, doi: https://doi.org/10.1145/2229012.2229035.

[81] L. Yu, “Mechanism design with partial verification and revelation principle,” *Autonomous Agents and Multi-Agent Systems*, vol. 22, no. 1, pp. 217–223, Jan. 2011, doi: https://doi.org/10.1007/s10458-010-9151-4.

[82] J. Ouaknine, A. Rabinovich, and J. Worrell, “Time-Bounded Verification,” *CONCUR 2009 - Concurrency Theory*. Springer, pp. 496–510, 2009, doi: https://doi.org/10.1007/978-3-642-04081-8_33.

[83] J. Ouaknine and J. Worrell, “Towards a Theory of Time-Bounded Verification,” *Automata, Languages and Programming*. Springer, pp. 22–37, 2010, doi: https://doi.org/10.1007/978-3-642-14162-1_3.

[84] N. Narodytska, “Formal Verification of Deep Neural Networks,” in *2018 Formal Methods in Computer Aided Design (FMCAD 2018)*, Austin, TX, USA, Oct. 2018, p. 1, doi: https://doi.org/10.23919/fmcad.2018.8603017.

[85] X. Sun, H. Khedr, and Y. Shoukry, “Formal verification of neural network controlled autonomous systems,” in *ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2019)*, Montreal, Quebec, Canada, Apr. 2019, pp. 147–156, doi: https://doi.org/10.1145/3302504.3311802.

[86] J. Grass and S. Zilberstein, “Anytime algorithm development tools,” *SIGART Bull.*, vol. 7, no. 2, pp. 20–27, Apr. 1996, doi: https://doi.org/10.1145/242587.242592.