

Rheinische-Westfälische Technische Hochschule Aachen  
Lehrstuhl für Informatik 2  
Software Modeling and Verification  
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

**Master's Thesis**

---

**Verifying Automotive C Code using  
Modern Software Model Checkers**

---

Lukas Westhofen

March 6, 2019

First Reviewer: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen  
Second Reviewer: Prof. Dr. Erika Ábrahám  
Advisor: Philipp Berger



## Abstract

In a safety-critical environment, guarantees over deployed programs play a vital role in a faultless execution. In the last decade, progress in formal methods has lead to attempts of integrating verification into industrial environments. This thesis presents an evaluation of academic software verifiers on two industrial-grade embedded C programs, provided by the Ford Motor Company. Through a transformation of the given functional requirements into formal specifications, a set of 208 verification tasks was created. On those, we examine result coverage and performance of CBMC, ESBMC, 2LS, SMACK, CPACHECKER, and ULTIMATEAUTOMIZER. The overall findings indicate that most verifiers are not yet optimized for industrial embedded code, and show room for improvements. We find CBMC to be a powerful verifier for identifying bugs, and enhancing it with  $k$ -induction represents a promising approach to verify such challenging embedded code. With the goal to ease the verification process, we also explore possible effects of a-priori static analysis approaches. By increasing total coverage, static slicing proves to be effective.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related work</b>	<b>4</b>
<b>3</b>	<b>Foundations</b>	<b>5</b>
3.1	Case studies . . . . .	5
3.1.1	Code . . . . .	5
3.1.2	Specifications . . . . .	9
3.2	Software verification techniques . . . . .	16
3.2.1	Satisfiability solving . . . . .	17
3.2.2	Bounded Model Checking . . . . .	18
3.2.3	Incremental Bounded Model Checking . . . . .	19
3.2.4	k-Induction . . . . .	19
3.2.5	Symbolic Execution . . . . .	23
3.2.6	Explicit Analysis . . . . .	24
3.2.7	Abstractions and CEGAR . . . . .	25
3.2.8	Predicate Abstraction . . . . .	27
3.2.9	Trace Abstraction . . . . .	28
3.3	C software verifiers . . . . .	29
3.3.1	CBMC . . . . .	32
3.3.2	CBMC Incremental . . . . .	33
3.3.3	ESBMC . . . . .	34
3.3.4	2LS . . . . .	34
3.3.5	SMACK . . . . .	36
3.3.6	CPAChecker . . . . .	37
3.3.7	UltimateAutomizer . . . . .	39
3.3.8	k-Induction . . . . .	39
3.4	Leverage points for improvement . . . . .	40
3.4.1	Code . . . . .	41
3.4.2	Verifiers . . . . .	44
<b>4</b>	<b>Experimental setup</b>	<b>46</b>
4.1	Experimental environment . . . . .	46
4.2	Code setup . . . . .	46
4.3	Verifier setup . . . . .	47
4.4	Examined leverage points . . . . .	50
<b>5</b>	<b>Results</b>	<b>52</b>
5.1	Overall results . . . . .	52
5.2	Verifier results . . . . .	56
5.2.1	CBMC . . . . .	56
5.2.2	CBMC Incremental . . . . .	58

5.2.3	ESBMC	59
5.2.4	2LS	60
5.2.5	SMACK	62
5.2.6	CPAChecker	62
5.2.7	UltimateAutomizer	64
5.2.8	k-Induction	65
5.3	Effect size of leverage points	66
5.3.1	Code	66
5.3.2	Verifiers	69
5.4	Influence of the verification property	71
5.5	Comparison to BTC	72
<b>6</b>	<b>Discussion</b>	<b>73</b>
6.1	Overall results	74
6.2	Verifier results	75
6.2.1	CBMC	76
6.2.2	CBMC Incremental	76
6.2.3	ESBMC	77
6.2.4	2LS	77
6.2.5	SMACK	77
6.2.6	CPAChecker	78
6.2.7	UltimateAutomizer	78
6.2.8	k-Induction	79
6.3	Effect size of leverage points	80
6.3.1	Code	80
6.3.2	Verifiers	81
6.4	Influence of the verification property	82
6.5	Comparison to BTC	83
<b>7</b>	<b>Future work</b>	<b>85</b>
<b>8</b>	<b>Conclusion</b>	<b>87</b>

## List of Figures

1	Model-based design process with MatLab Simulink . . . . .	5
2	Embedded-style C program template . . . . .	7
3	Transformation from a requirement to a specification . . . . .	10
4	Code implementation of a specification . . . . .	12
5	Variable impact graph of ECC and DSR . . . . .	15
6	Verification technique overview . . . . .	17
7	Bounded model checking example programs . . . . .	19
8	$k$ -induction transformation . . . . .	21
9	Illustration of $k$ -induction . . . . .	21
10	$k$ -induction example program . . . . .	22
11	Symbolic execution example . . . . .	24
12	Over-approximation and abstraction example . . . . .	25
13	CEGAR procedure . . . . .	26
14	CEGAR iteration . . . . .	27
15	Automaton-based CEGAR procedure . . . . .	29
16	State space representations of the verifiers . . . . .	32
17	Slicing example . . . . .	42
18	Value analysis example . . . . .	43
19	Variable moving example . . . . .	44
20	Quantile plots for the overall results . . . . .	53
21	Quantile plots for ECC and DSR . . . . .	54
22	Overall results for each verifier . . . . .	54
23	Overall results for each verifier for ECC and DSR . . . . .	55
24	Euler's diagram of the distribution of verifier results . . . . .	55
25	CPU-times of CBMC against ESBMC and 2LS . . . . .	57
26	CPU-times of CBMC . . . . .	58
27	CPU-times of CBMC against CBMC INCREMENTAL . . . . .	58
28	CPU-times of CBMC INCREMENTAL . . . . .	59
29	CPU-times of ESBMC against 2LS, CPACHECKER, and K-INDUCTION . . . . .	60
30	CPU-times of ESBMC . . . . .	60
31	CPU-times of 2LS against K-INDUCTION and CPACHECKER . . . . .	61
32	CPU-times of 2LS . . . . .	61
33	CPU-times of CPACHECKER against its analyses and ULTIMATEAUTOMIZER . . . . .	63
34	CPU-times of CPACHECKER . . . . .	63
35	Euler's diagram of the distribution of CPACHECKER analyses results . . . . .	64
36	CPU-times of ULTIMATEAUTOMIZER . . . . .	65
37	CPU-times of K-INDUCTION . . . . .	66
38	Relative slice sizes for ECC and DSR . . . . .	69
39	Overall requirement results with BTC . . . . .	73
40	ECC and DSR requirement results with BTC . . . . .	73
41	Program with undefined behavior . . . . .	76

## List of Tables

1	Code metrics of ECC and DSR. . . . .	8
2	Example requirement of an FRD . . . . .	9
3	Specification analysis of ECC and DSR. . . . .	13
4	Category weights for the relative verifier score . . . . .	31
5	Relative verifier scores . . . . .	31
6	CPA operators . . . . .	38
7	SMT-solver support of the verifiers . . . . .	45
8	Final scores of the verifiers . . . . .	52
9	Verifier memory usage . . . . .	56
10	Counterexample depths for CBMC . . . . .	57
11	Proof depths of K-INDUCTION . . . . .	65
12	Significances and effect sizes of the code leverage points . . . . .	67
13	Amount of CPU-time spent in an SMT-solver . . . . .	70
14	Significances and effect sizes of the verifier leverage points . . . . .	70
15	Unverified tasks per category . . . . .	71
16	Significances and mean difference between unverified and verified tasks . .	72



# 1 Introduction

Ever since the first implementation of a functionally complete computer by Konrad Zuse in 1941, digitization of previously analogous tasks has advanced steadily. Initially only used for specialized purposes, computers have continued to revolutionize our everyday life over and over again. As the ubiquity of digital devices increases, the complexity of the employed hard- and software rises along. Moreover, engineers and researchers strive to create even more sophisticated systems. Due to this development, their analysis has played an important role ever after Alan Turing laid the foundations of a formal computational machine model. Early in computer science history, the natural question arose whether a machine can automatically decide if a program adheres to a certain behavior.

Despite the fact that such problems can not be solved in general, this central question gave birth to field of software verification. In his 1948 paper, Turing is the first to present the idea of assertions for checking a large routine [Turing 1948], still involving manual labor. It was only in 1967 that the scientific community started exploring the possibility of automated correctness proofs for computer programs [Floyd 1967]. Tony Hoare's introduction of axiomatic semantics for programming languages gave birth to a framework for logical reasoning on programs [Hoare 1969].

Although this challenge proved difficult at first, automated software verification has taken large steps towards maturity and applicability since then. For example, the concepts of temporal logic, abstract interpretation, and predicate transformer semantics were introduced in the 1970s. Clarke, Emmerson, Queille, and Sifakis then pioneered the field of model checking in the following decade [Emerson and Clarke 1980] [Queille and Sifakis 1982]. Despite those advances, the ever growing complexity of programming languages and their applications raise the bar for a transfer of the theoretical concepts to the industrial community, although its necessity has been pointed out early on:

*“The next rocket to go astray because of a programming language error may not be an exploratory space vehicle on a harmless trip to Venus. It may be a nuclear warhead exploding over one of our own cities.”*

– Tony Hoare, 1981

Luckily, this prediction of a nuclear scale programming error has not (yet) come true. But since then, the usage of inherently unsafe programming languages has risen: in some cases, engineers find themselves stuck with language tools not being able to give guarantees over their designs and implementations. From the early 2000s on, the hard- and software industry realized the importance and capabilities of formal verification:

*“Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.”*

– Bill Gates, 2002

Shortly thereafter, Tony Hoare proposed a *grand challenge* for a verifying compiler in

2003, with the goal of collectively working towards increasing “*confidence in the structural soundness of the system*” [Hoare 2003].

15 years later, we observe a wide variety of software verifiers for a multitude of programming languages being developed by various research groups worldwide. Although they provide an integral part in a reliable end-product, in order to maximize robustness, multiple analysis methods are often employed. As Hoare noted in his proposal, a verifier “*will work in combination with other program development and testing tools*”. Thus, verifiers do not compete, but rather complement traditional testing methods as well as sound design and implementation principles. Although testing is able to reduce the number of bugs and increase confidence in the correct behavior, verification provides guarantees over the program by an exhaustive search of the state space. As it is all too often the case, proving correctness is a far more demanding task than identifying erroneous behavior. As such, applicability of formal methods has been restricted to small examples for most of its history, although starting in the last decades, we find multiple larger projects being tackled, including:

- The *Ariane 5* rocket, where after its catastrophic failure in 1996, an effort was made to verify all critical components [Lacan et al. 1998].
- The Microsoft *SLAM* project, which is concerned with guarantees over the reliability of software interfaces, and was applied in the Static Device Driver tool [Ball et al. 2004].
- *Astrée*, a static analyzer, was used on the Airbus A380 [Souyris 2004].
- The *Linux Device Driver* project, which provides a tool set for static source code analysis of Linux device drivers [Khoroshilov et al. 2010].

As those examples demonstrate, formal methods have been successfully deployed to industrial-scale use cases, and exhibit promising results. Drawing on such applications, this work examines the applicability of modern software verifiers on automotive embedded programs.

In the automotive industry, reliability and robustness of the deployed software is of utmost importance. This is reflected by the functional safety standard for road vehicles, ISO 26262, and the therein defined Automotive Safety Integrity Level, published in 2011. Their main target is the reduction of residual risks by posing requirements on the safety of a system. This standard covers all phases of a system’s development process, such as specification, design, implementation, verification, validation, and configuration. Adhering to such high standards of reliability is not an easy task – most automotive systems are organized in multiple subsystems that contain hundreds to thousands of semantic units. Moreover, features such as the communication of systems through global variables further increase the complexity of the implementations. Manual analysis of those involved and interdependent systems is an exercise in futility. Moreover, even after product tests, an average of 15 to 50 defects per 1000 lines of code can be found in a typical industrial application [McConnell 2004].

Due to this, the automotive industry has acknowledged the need for automated verification. This comes to no surprise, as a high economic incentive is given: the cost of software defects are estimated to increase drastically the later they are discovered [Boehm and Papaccio 1988]. In 2001, Holzmann examined the economics of software verification. He concludes that “*the question is perhaps not ‘what is the justification for using software verification techniques in software development’, but ‘what would be the justification for not doing so?’*” [Holzmann 2001]. Similarly, NASA found that it is possible to achieve extremely low residual defect rates [Dvorak 2009] – although this comes at a cost. For example, NASA has spent roughly 35% of its 200 million Dollar Space Shuttle software budget on verification and validation<sup>1</sup>. For companies in a highly competitive market, such costs are often economically infeasible. For them, automated approaches that make use of already existing information, such as functional requirements, can represent a more appropriate means.

Based on the previously presented reasoning, we state the leading question of this thesis:

*Can modern software verifiers be applied to today’s embedded automotive C code, and if so, to which extent? If there is room for improvement, what are possible ways to optimize their usage towards greater coverage and efficiency?*

Thus, we aim to find a suitable approach for the integration of software verification into the automotive development process by examining possible strategies. Section 3 presents fundamental information on our procedure.

To investigate possible answers to the leading question, the Ford Motor Company has provided us with two case studies that are representative of the previously described automotive systems. Both originated from a model-driven development process and contain challenging features, such as floating point operations, a large amount of global variables, and pointer arithmetic. The actual verification is performed on code-level as opposed to model-level, meaning that the modeled system was exported to C code and then verified. Alongside both implementations, we have been supplied with descriptions of the functional requirements, to which the systems shall adhere. We formalized those requirements, creating a set of 208 verification tasks for the examined software verifiers. The case studies and functional requirements are introduced in Section 3.1.

The key point of this thesis is the observation of the verifiers’ result coverage and performance on the verification tasks. For this, we selected a wide range of software verifiers that employ a multitude of techniques. We will introduce the basic concepts of their approaches and give a short overview on their history and development in Section 3.2 respectively 3.3.

With the goal to further improve efficacy of software verification for the given use case, we present various leverage points on multiple levels in Section 3.4.

We performed an experimental benchmark of the selected set of verifiers on all tasks, described in Section 4. The results are presented in Section 5 and discussed in Section 6.

---

<sup>1</sup><https://history.nasa.gov/computers/Ch4-5.html>

## 2 Related work

The most complete evaluation of the state of today’s academic verification tools is the annual *Competition on Software Verification* (SV-COMP), hosted by the TACAS conference [Beyer 2017]. In contrast to our work, where we examine industrial automotive C code, the SV-COMP focuses on a wide range of programs, from smaller academically oriented tasks, parallelism, and memory safety up to real software systems.

As most of the industry’s code is not publicly available, there exist only few academic evaluations thereof. The industrial tool BTC validator and its underlying verifiers are tested on a large suite of industrial programs, as presented in [Schrammel et al. 2015] and [Schrammel et al. 2017]. Moreover, an academic case study on embedded C code was conducted on the verifier CBMC [Schlich and Kowalewski 2009]. To our knowledge, an extensive evaluation of the current state of academic software verifiers on industrial-grade embedded programs has not yet been conducted.

Although not a complete benchmark, we observe various reports and evaluations on applications of formal methods in practice, such as in a martian rover [Brat et al. 2004], CERN [Adiego et al. 2014], and Facebook [Calcagno et al. 2015]. [Botaschanjan et al. 2005] presents a verification approach for subsystems of automotive C code.

This work examines code that originated from model driven development. Venkatesh et al. argue that such an approach is preferable over testing the model directly [Venkatesh et al. 2012]. Nellen et al. verified MatLab Simulink models, similar to the two case studies at hand [Nellen et al. 2018]. Another extensive study of the Simulink Design Verifier was conducted on an monitoring system for aero-engines [Bennion and Habli 2014]. Both case studies presented in this work were previously examined with an industrial verification suite, BTC [Berger et al. 2018].

Besides formal verification, there also exist testing approaches such as concolic testing [Sen et al. 2005] and symbolic execution [Cadar et al. 2008]. Beyer et al. evaluated and compared such test generation approaches to model checkers [Beyer and Lemberger 2017], concluding that the latter are able to find more bugs in a shorter amount of time.

Besides examining the verifier performance, we are also concerned with the formalization of requirements. This is an active field of study in the areas of domain ontologies, knowledge representation, and natural language processing. As an example, [Böschchen et al. 2016] presents the Requirement Quality Suite, a tool chain that supports and semi-automates the mentioned process. With [Ilieva and Ormandjieva 2005], there also exists an automated approach to this problem, recognizing the pitfall of ambiguity and misinterpretation in functional requirements. [Chomicz et al. 2017] presents an approach based on controlled natural languages, where the dangers of complex and ambiguous documentations in the automotive industry are mitigated by introducing semi-formalisms.

Finally, we present our implementation of a configurable  $k$ -induction tool that runs on top of any bounded model checker. Comparable to our development is DEPTHK, but its back-end verifier is not interchangeable and fixed to ESBMC [Rocha et al. 2017].

### 3 Foundations

In the following sections, we present foundations and techniques necessary to interpret this work’s results. For this, we introduce and analyze the case studies, present the selected verifiers along with their technical approaches, and lastly introduce possible leverage points to improve on verification results.

#### 3.1 Case studies

The verification experiment in this analysis is applied to two case studies from the automotive domain that were made available by our research partner, Ford: An *Electronic Clutch Control* (ECC) and a *Driveline State Request* (DSR). Distinguishing features of both will be outlined. Both share a general structure and features as well as a common library code base, all of which we discuss in the next subsection.

Each implementation has previously defined specifications to adhere to. They are given in the form of a *Functional Requirements Document* (FRD), and described in the second half of this section.

##### 3.1.1 Code

A model-based design approach was chosen for both case studies. Such a development process is typical for the automotive industry [Friedman 2006]. Starting with a given set of requirements over the set of input and output variables, a model was created using MatLab Simulink. In this software, engineers are able to model components in a graphical setting. It also enables simulation which plays an imperative role in *rapid prototyping* and *test as you go* [Friedman 2006].

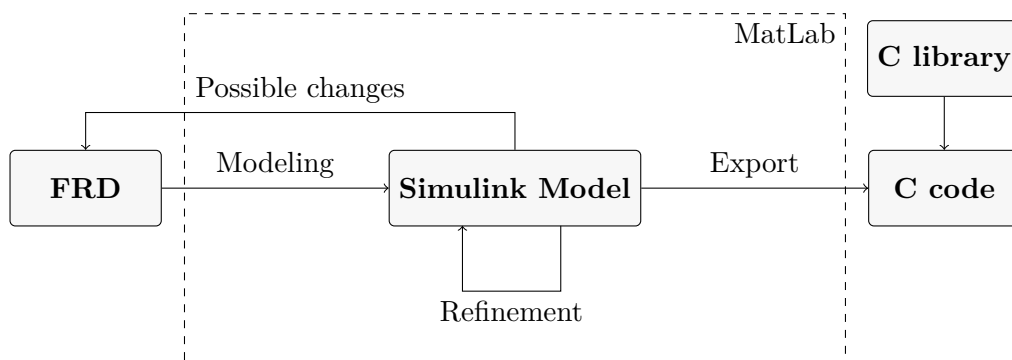


Figure 1: A model-based design process based on MatLab Simulink.

During the process depicted in Figure 1, the engineers are able to refine and change the model based on the simulation. Finally, the model is then exported to executable C code using the MatLab export functionality. Model-based design approaches have become

widespread in the last two decades, as this lessens the expertise for actual coding and allows skilled engineers focus on the task in a timely manner [Schätz et al. 2002]. In contrast to building a prototype first and testing the implementation afterwards, being able to simulate models during development can improve the implementation process. Both case studies access a common, hand-written C library, which implements various commonly used utility functions such as interpolation, data lookups, and mathematical operations.

Compared to classic code development, an executable design environment can lead to different verification approaches: Firstly, one can examine the model directly. Alternatively, its implementation can be analyzed. While approaches for the first method exist, e.g. through the Simulink Design Verifier [Sim 2007] [Nellen et al. 2018] or model checkers such as SPIN [Leitner and Leue 2008], this work focuses on the latter. As the exported code can include features such as pointer arithmetic and dynamic memory allocation, abstraction may be lost in the transformation. Despite this, we justify the approach by finding errors possibly introduced in the translation step. Those faults can arise due to bugs in the transformation code or misinterpretations of the model semantics. Additionally, it was found that verifying code directly is agnostic of the actual model semantics, meaning that one verification tool can be used for multiple sources of models [Venkatesh et al. 2012]. This includes advantages such as reducing possible errors in the verification tool chain through simplification as well as choosing from a wide range of verifiers.

Both case studies at hand are examined in an open-loop setting, meaning that we do not impose restrictions on the controlled environment. The examined code falls in the category of *embedded-style* programs; it adheres to the template presented in Figure 2.

For the verification task itself, the function `updateVariables()` is added to the code right before each `step()` call. This function sets each input variable to a non-deterministic value in order to emulate the open-loop setting.

In the following, we present analyses and metrics on the code. This allows for a more precise selection of verifiers since we can match their strong suits to this particular use case. Furthermore, connecting code metrics to the verification results enables a more detailed evaluation later on.

**ECC:** Implements a control system for an electronic clutch. In a car, the clutch is responsible for connecting the engine and gearbox and thus allowing the transfer of mechanical work from one to the other. The electronic clutch is a recent development of the automotive industry with the goal of improving the car’s efficiency. Here, the actual coupling between the two shafts is not done manually but by an electronic actuator. This enables a digital access to the clutch’s state, for which ECC implements the necessary operations. This includes features such as torque calculations for opening and closing the clutch.

**DSR:** Implements a query system for the current state of the driveline. Also called powertrain, it encompasses all components of the car responsible for generating power

```

1 // Global variables can be declared here.
2
3 void initialize() {
4     // Initializes global variables.
5 }
6
7 void step() {
8     // Monolithic code for one discrete step.
9     // Each step is bounded in its run time.
10    // Local variables can be declared here.
11 }
12
13 void main() {
14     // Entry point.
15     initialize();
16     // Executes the step indefinitely.
17     while(1) {
18         step();
19     }
20 }

```

Figure 2: A template for embedded-style C programs.

and delivering it to the road’s surface. This includes, among others, the engine, drive shaft, and wheels. DSR signals the state of the connection between those components – they can be transmitting no, only positive or all torque – and implements their transitions. On top, it features an idle coasting mode. Here, if a situation arises in which acceleration is not needed, the car’s powertrain is uncoupled from the engine to reduce fuel usage.

**Code analysis:** From the Simulink Models, generated from a few thousand blocks, around 2500 and 1300 source lines of C code were extracted for ECC respectively DSR. Table 1 presents the metrics collected on both case studies.

Constants are used to account for configurability, i.e. they represent parameters of the model that can be changed for different types of applications. The configurable state space consists of 274 and 77 constants, for ECC and DSR respectively. Most of them are of the `float` type, sometimes in an a fixed-length array. Additionally, both case studies contain pointers to constant data (e.g. `const void*`). With a couple of hundred variables, globals are heavily employed. They are used for exchanging data with other compilation units. Here, the `char` type is most prevalent, taking up around three quarters of the variable count. `float` variables make up the remaining quarter.

The number of operations that are present in the call graph total to around 10 000 and 5 000 for ECC and DSR. While linear arithmetic is most prominent, we also observe a large amount of multiplication and division, possibly on non-constant variables. Challenges for software verifiers rise along with the complexity of operators used. Pointer and floating point arithmetic, as well as bit-wise operations impose challenges. These case

Table 1: Code metrics of ECC and DSR.

Metric	ECC	DSR
<i>Global constants</i>	274	77
char	8	12
char []	0	[12,32] 2
float	77	35
float []	[2-7] 4	[6-12] 9
float*	1	1
void*	184	18
<i>Global variables</i>	775	273
char	595	199
char []	0	[16-32] 3
float	110	46
float []	[2-4] 70	[4-10] 25
<i>Operations</i>	10096	5232
Addition/subtraction	346	133
Multiplication/division	253	52
Bit-precise operations	191	65
Pointer dereferences	180	83
<i>Complexity</i>		
Source lines of code	2517	1354
Cyclomatic complexity	268	213
Halstead length	15498	7890
Halstead volume	150973	71165
Halstead difficulty	398	378

studies employ a variety of bit-precise operations such as `»`, `&`, and `|`. Such operators force the underlying solvers to model the variable bit for bit, and thus remove abstraction from the program. With around 180 and 80 dereferences, a noticeable amount of pointer arithmetic is used in the programs.

The cyclomatic complexity indicates the number of possible paths through the program’s control flow graph, without taking loops multiple times – this is called linearly independent paths [McCabe 1976]. It can be calculated using the formula  $|E| - |V| - 2 \cdot |C|$  where  $E$  is the set of edges,  $V$  the set of vertices, and  $C$  the set of connected components. The main loop is thus ignored in this calculation, meaning that there are 268 (ECC) and 213 (DSR) linearly independent paths in one single execution step.

We also present the Halstead complexity measures, originally introduced by Howard Halstead in 1977 [Halstead and Others 1977]. The *program length* consists of the sum

of all operators and operands. This metric evaluates to lengths of 15 000 (ECC) and 7 500 (DSR). The *program volume* is calculated based on the length multiplied by the logarithm of the amount of distinct operators and operands. Intuitively, it represents the storage size of a programs, in bits. Hence, we observe approximate sizes of 150 kbit and 75 kbit. Finally, the difficulty is proportional to the number of redundant operands and the amount of total operators. It relates to the complexity of understanding and writing such code. With both case studies being in the high three-hundreds, we observe a comparable difficulty between both case studies. It has to be noted that all Halstead measures are typically applied to human-written programs, and in this case are only used to compare both programs against each other.

### 3.1.2 Specifications

The verification task presents itself through specifications in an FRD, one for each case study. Those documents contain a listing of requirements, where each can pose restrictions over the in- and output variables, as well as the internal semantics and implementation details of the program. A fictional exemplary requirement is presented in Table 2.

Table 2: An example of a requirement present in an FRD.

State <b>S</b> influences variable <b>X</b>	REQ-1
If <b>&lt;S&gt;</b> transitions to 1 or 2, <b>&lt;X&gt;</b> shall be set to <b>TRUE</b> and a timer shall be initialized. <b>X</b> shall be <b>TRUE</b> until the timer expires after ( <b>cal: X_steps</b> ) steps.	

Although requirements form semantically contained units, they can often be significantly longer than the example, arguing about multiple connections between variables and algorithms.

As noticeable in the example, the specification is given in a textual format without any formalisms and specified syntax or semantics. This leads to the necessity of transforming a statement from natural language to a formal specification with the goal of feeding it to a verifier. Ultimately, the specifications will be represented as a piece of C code that checks for violations of the requirement. Independent of the actual transformation format, this step is highly susceptible to the transformers interpretation of the requirement author's intentions.

**Specification process:** In our case, we used the *Pattern Based Specification Tool* which gives a graphical interface for requirement formalization and was developed by Johanna Nellen and Philipp Berger during their work with Ford [Nellen et al. 2018] [Berger et al. 2018].

An overview of the transformation process is given in Figure 3. In the following, we go through the illustrated process by formalizing the example requirement of Table 2.

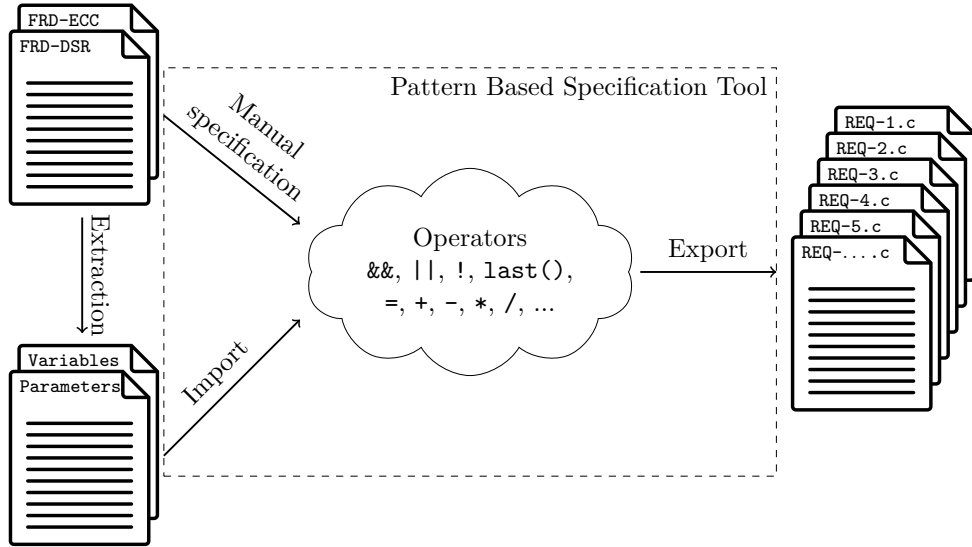


Figure 3: A transformation from natural language requirements to code specifications.

Firstly, based on the FRDs, variables and their types are fed into the tool. The specifications are later able to argue over those in- and output variables. In our example, this would encompass  $X$  of type `boolean` and  $S$  of type `unsigned int`.

A formal specification is then extracted from each requirement. We specify those by means of a fragment of linear time logic (LTL) over propositional formulae containing typical C arithmetic, logical operators, and arrays. Further, a restricted temporal operator is added: `last(x)`, which returns the value of the variable  $x$  in the previous main loop iteration. Any specification can either be defined to hold globally –  $\bigcirc$  is prepended – or initially – no temporal operator is needed. The tool currently supports two formal patterns providing a subset of LTL:

- **Invariant:** One propositional formula  $\varphi$ . If specified globally, i.e.  $\bigcirc\varphi$ , it denotes that  $\varphi$  is valid after (and before) every loop iteration. This is also called a *loop invariant*.
- **Trigger-Response:** Two propositional formulae  $\varphi, \psi$ , and three delays  $d_1, d_2, d_3$  such that  $((\bigwedge_{i=0}^{d_1-1} \bigcirc^i \varphi) \implies (\bigcirc^{d_1+d_2-1} (\bigwedge_{i=0}^{d_1+d_2+d_3-1} \bigcirc^i \psi)))$  holds. The delays are given as number of loop iterations, although the tool also allows a declaration in time units which can be converted to steps. Intuitively, it holds iff. if  $\varphi$  is valid for  $d_1$  steps then, after a delay of  $d_2$  steps,  $\psi$  has to hold for  $d_3$  steps.

During the specification phase, requirements possibly need to be split up into multiple specifications. This arises due to semantical differences, e.g. in case a requirement contains an initial and a global part. Sometimes, formal specifications become large enough that one monolithic specification would lead to impediments during the verification analysis – If a counterexample is found in one subpart, one can not analyze whether the rest

of the specification is actually valid, which may be the case. Thus, we split requirements into semantically contained sub-specifications if necessary. The exemplary requirement is specified by employing both a global invariant and trigger response. The invariant for setting  $X$  is specified as follows:

$$\varphi = ((\neg(\text{last}(\mathbf{S}) = 1) \wedge (\mathbf{S} = 1)) \vee (\neg(\text{last}(\mathbf{S}) = 2) \wedge (\mathbf{S} = 2))) \implies (\mathbf{X} = 1)$$

To emulate the timer, we use a trigger-response pattern with the durations  $d_1 = 1$ ,  $d_2 = 0$ ,  $d_3 = X\_steps$  and the following formulae:

$$\begin{aligned} \varphi &= (\neg(\text{last}(\mathbf{S}) = 1) \wedge (\mathbf{S} = 1)) \vee (\neg(\text{last}(\mathbf{S}) = 2) \wedge (\mathbf{S} = 2)) \\ \psi &= (\mathbf{X} = 0) \end{aligned}$$

Once such propositional formulae are specified in their respective patterns, the requirements can be exported to C files.

This last step will produce the code listed in Figure 4. Each file represents one specification in the general form of `if (!property) { __VERIFIER_error(); }`. In the case of the `last()` operator and trigger-response patterns, additional variables containing a look-back are added and updated in the main loop accordingly.

The first fragment shows the formalization for setting  $X$  on the state transition. This involves the look-back variable `S_last_1` which is updated at the very end of each iteration. The second fragment displays the more involved part of checking the trigger-response pattern: here, `P_` and `Q` are used to store the history of the premise  $\varphi$  and the conclusion  $\psi$ . For `Q` it is sufficient to count how far the current streak of  $\psi$  satisfactions reaches. For  $\varphi$  on the other hand, one needs to find a value at a certain previous time point, meaning that each value of  $\varphi$  needs to be stored for each iteration. This is done by means of the array `P_` which implements an `X_steps` long look-back.

During the course of this work, the Pattern Based Specification Tool was enhanced by the following features to be applicable to the specific case studies:

- **Array access:** Support for the array access operator `[]` was added and its export to C code was implemented.
- **Variable durations:** Durations of trigger-response patterns can now be defined by means of some variable, in contrast to being restricted to an integer value.
- **Floating point comparisons:** When exporting to C code, any comparison between two floats now accounts for small imprecisions that may arise due to floating point arithmetic.
- **Batch export:** This feature allows to export all requirements with one command, but outputs each to a single file instead of one large file containing all requirements.

**Specification analysis:** In the following, we present measures to characterize the given verification tasks at hand. Most of the analyses were not executed on the *requirements*

```

1 unsigned int S_last_1;
2 while (1) {
3     updateVariables(); step();
4     if (!(!(S_last_1==1 && S==1) || !(S_last_1==2 && S==2)) || (X==1))
5         __VERIFIER_error(); // Checks for property violation.
6     S_last_1 = S; // Stores the iteration's value to access it in the next.
7 }

1 // Counter for the number of loop iterations.
2 unsigned int counter = 1;
3 unsigned int hasOverflown = 0;
4 // Stores the history of the premise's truth values.
5 unsigned short P_[X_steps];
6 memset(P_, 0, X_steps);
7 // Stores the length of the conclusion's last truth streak.
8 unsigned int Q = 0;
9 while (1) {
10    updateVariables(); step();
11    // Shifts to the left to make room for the new premise's truth value.
12    for (int i = 0; i < X_steps - 1; i++) P_[i] = P_[i + 1];
13    // Stores the new premise's truth value.
14    P_[X_steps - 1] = (!(S_last_1==1 && S==1) || !(S_last_1==2 && S==2));
15    // If conclusion holds, updates its truth streak, resets it otherwise.
16    if (X == 1) Q++;
17    else Q = 0;
18    // First, checks if we can actually look far enough into the history.
19    if (counter >= X_steps || hasOverflown)
20        // Checks for violation of "premise implies conclusion".
21        if (!(P_[0] || (Q >= X_steps)))
22            __VERIFIER_error();
23    S_last_1 = S;
24    // Updates counter or overflow, if we reached sufficient iterations.
25    if (counter > X_steps) hasOverflown = 1;
26    else counter++;
27 }

```

Figure 4: The final code resulting from the two specifications of Table 2.

themselves but rather on the formalized *specifications*. As noted previously, they represent the actual verification tasks, where one requirement may be split up into multiple specifications.

Table 3 shows the metrics collected on both case studies. ECC and DSR contain 80 and 46 requirements, respectively, most of which we were able to formalize into verification tasks. Some requirements were not formalizable. For those, *not implemented* denotes that the variables of the specified feature were not yet added to the actual code due to the feature being not yet implemented. *Underspecification* occurs when the natural-language requirement is vague in its denotation and allows for various interpretations that would lead to different verification outcomes. For example, in ECC we observed

Table 3: Specification analysis of ECC and DSR.

<b>Metric</b>	<b>ECC</b>	<b>DSR</b>
<i>Requirements</i>	<i>80</i>	<i>46</i>
Fully specified	54	36
Partially specified – Underspecification	6	1
Partially specified – Not implemented	8	2
Not specified – Underspecification	0	1
Not specified – Not implemented	12	6
<i>Specifications</i>	<i>136</i>	<i>72</i>
Initial invariants	10	2
Global invariants	125	68
Initial trigger-response	0	0
Global trigger-response	1	3
Specifications with feature...		
Automaton initial condition	2	6
Automaton transition	77	34
Hysteresis verification	0	7
Hysteresis access	5	24
Array access	0	23
Floating point comparison	38	13
Bit-precise operations	1	0
<code>last()</code>	83	57
Direct global variable access (mean±SD)	4.5 ± 2.7	3.8 ± 1.5
float	2.3 ± 2.0	0.4 ± 0.8
char	2.2 ± 1.7	3.4 ± 1.8
Direct global constant access (mean±SD)	2.0 ± 1.6	0.7 ± 0.9
float	1.0 ± 1.4	0.4 ± 0.8
char	0.9 ± 1.0	0.3 ± 0.5
Indirect global variable access (mean±SD)	27.8 ± 9.5	12.7 ± 8.7
float	19.9 ± 6.8	2.7 ± 2.1
char	7.9 ± 2.9	10.0 ± 6.8
Indirect global constant access (mean±SD)	25.7 ± 11.6	7.2 ± 7.7
float	20.8 ± 8.3	4.6 ± 3.9
char	4.9 ± 4.1	2.6 ± 3.9

requirements arguing over a “2nd order low pass filter” or “rate limiting” without specifying the required parameters and implementation details of such a feature. For a detailed description of all issues in the underlying FRDs, we refer to [Berger et al. 2018].

The formalization process led from 80 and 46 requirements to 136 and 72 specifications for ECC and DSR, respectively. This blow-up can be accounted mostly by automaton requirements, where each transition was modeled as a single specification. In total, automaton formalizations make up 58% and 56% of the ECC and DSR specifications. The case studies additionally contain hysteresis features, where variable values are “smoothened” to reduce sensitivity to small changes around a given threshold. Those smoothened hysteresis variables have to be verified separately, which accounts for 7 of the 72 DSR specifications. Arrays are accessed in only a third of the DSR specifications. Lastly, the `last()` operator is used in 61% and 79% of the ECC and DSR specifications.

The requirements should – in theory – comprehend the complete behavior of the code and thus give us an estimate of the interdependency of specifications and variables. Considering this, we constructed a *variable impact graph* to measure the complexity of the specifications. For the set of specifications  $Spec$  and their variables  $Var$ , this graph’s node set is  $V = Spec \cup Var$ . We then add an edge  $(v_1, v_2)$  to  $E \subseteq V \times V$  in the following cases:

- $v_1 \in Spec, v_2 \in Var$ , and specification  $v_1$  argues over variable  $v_2$ .
- $v_1, v_2 \in Var$  and there exists  $s \in Spec$  s.t. in  $s$ ,  $v_2$  has to be accessed to set the value of  $v_1$ .

For our exemplary specifications of Table 2, the graph consists of nodes  $\{REQ-1-Inv, REQ-1-TR, X, S, X\_steps\}$  and edges  $\{(REQ-1-Inv, X), (REQ-1-Inv, S), (REQ-1-TR, X), (REQ-1-TR, S), (REQ-1-TR, X\_steps), (X, S), (X, X\_steps)\}$ .

We extracted such a graph semi-automatically from the formalized specifications of the case studies. Figure 5 shows a visualization of said graphs for both case studies. In these graphs, one can now find multiple measures:

- The variables that a specification accesses directly, by fetching the neighbors of the corresponding node.
- The variables that have to be accessed in order to have a definite value for a selected variable. This can be found by computing the reachability set for each variable node. Note that this is a possible over-approximation, as the order of reading and writing is not stored in the impact graph.
- The variables on whose values a specification indirectly depends on. This accounts for interdependency of the variables directly accessed in the selected specification. This procedure is similar to the previous, but executed for each variable neighbor of the specification.

Ultimately, we denote a specification to be more complex if it depends on a larger set of variables, as this leads to larger program slices as well as the need for finer abstractions. This interdependency is stated in Table 3, where the mean and standard deviation of

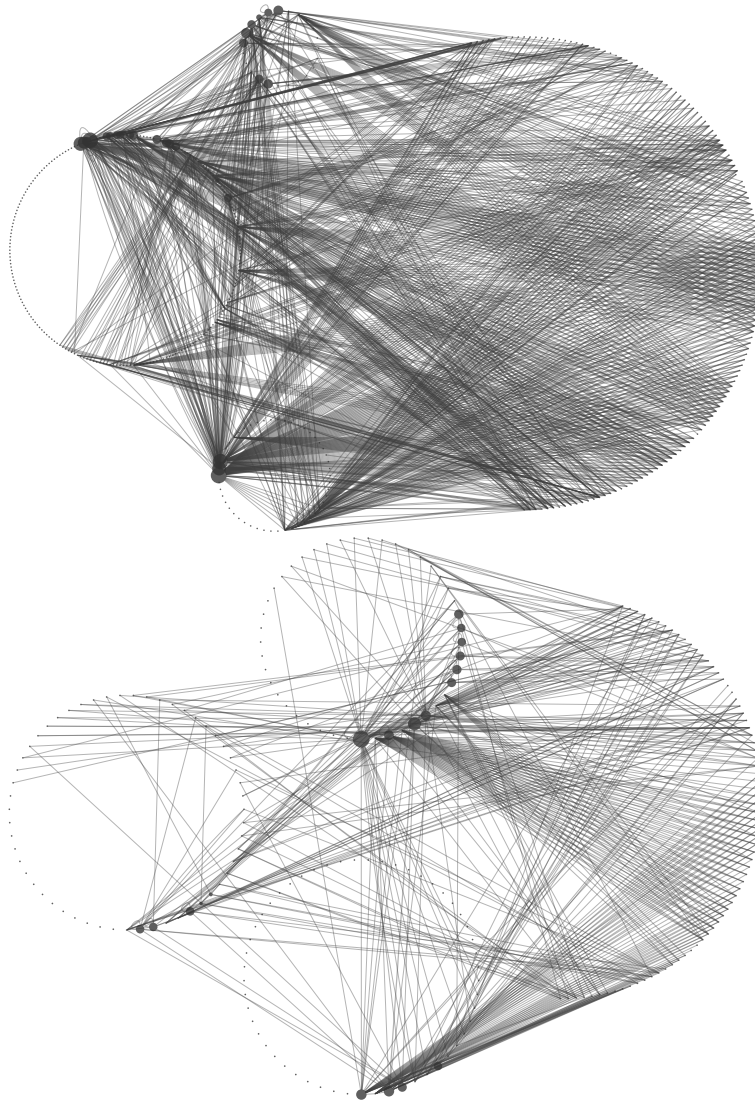


Figure 5: The variable impact graph of the ECC (upper graph) and DSR (lower graph) case studies. Larger nodes have a higher betweenness-centrality value. For each case study, the right half-circle contains the requirement nodes, where the three left full-circles represent the variables. The upper most circle contains boolean, the middle one floating point and the lowest integer variables.

the global variable and constant access is given for each case study. With around four variables, most requirements argue over a small set of globals. This picture changes if the dependencies between requirements are accounted for: then, the mean increases to around 28 variables for ECC and 13 variables for DSR, with the standard deviation exhibiting a quite large variability. This implies the existence of requirements which are inherently heavily dependent on the global program state, whereas others can be verified

by observing a small subset of variables.

### 3.2 Software verification techniques

As research on software verification has increased in the last decades, so has the number and sophistication of verification approaches. This section gives an overview of the state-of-the-art techniques employed by our selection of verifiers. Those approaches define the strengths and weaknesses of the verification process and ultimately decide which tasks are solvable for a given verifier.

In this case study, all verifiers reduce their approach to a satisfiability problem, which poses the question whether a given formula has a solution or not. While this problem is NP-complete in theory [Cook 1971], today’s algorithms and solvers can tackle most real-world problems in an efficient manner. Although there are verification approaches not based on satisfiability theories, the modern verifiers competing in the SV-COMP made the decision to incorporate a solver back-end, although to a varying degree.

All verification techniques can be classified by their approach to the approximation of the program semantics, as for most programs it is neither feasible nor possible to examine all program paths. Typically, a technique falls into one of three categories:

- **Under-approximation:** Examine only selected paths or path prefixes of the program. Can give counterexamples, but fails to provide proofs, as not all program parts were examined for validity. For example, unrolling each loop of a program once and examining the resulting state space under-approximates the full state space, if the loop’s bound is greater than one.
- **Exact:** Examine the program exactly as its semantics specify. As an example, a verifier can try to construct the complete state space of a program, and then run its analysis on the resulting states.
- **Over-approximation:** Examine more paths than the program semantics specify. Can deliver proofs, but can not guarantee that a found counterexample is present in the actual program. Exemplary, multiple concrete states can be merged into a single abstract one, in order to reduce the state space.

Those strategies can also be combined in order to achieve a more powerful verifier: One could, for example, find a counterexample with an over-approximating technique first, and then verify its validity using a simulation under exact semantics to exclude what is called *spurious counterexamples*.

Overall, Figure 6 gives a summary on the various techniques presented in the upcoming subsections. Whereas bounded model checking under-approximate the program’s semantics, symbolic execution in a standard setting and explicit value analysis try an exact interpretation. In case this is not feasible due to extremely large state spaces, one can resort to  $k$ -induction or predicate and trace abstraction.

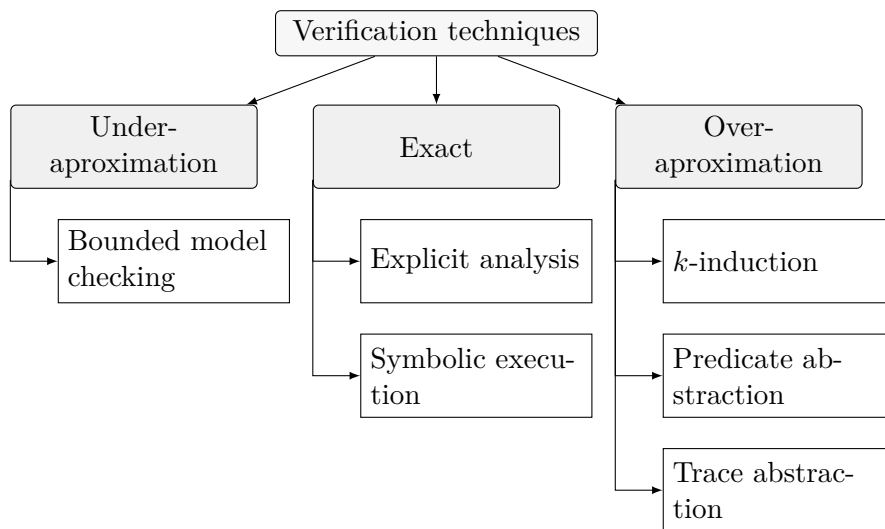


Figure 6: An overview on the evaluated techniques employed by various verifiers used in this study.

### 3.2.1 Satisfiability solving

All verification techniques described in the upcoming sections rely on a satisfiability solver to a certain extent. In today’s verifiers, mostly solvers for the *Satisfiability Modulo Theories* (SMT) are prevalent, although sometimes *Boolean Satisfiability Problem* (SAT) solvers can be observed.

**Definition 1. SAT-problem.** *Given a boolean formula  $\varphi$ , decide whether there exists a satisfiable variable assignment for  $\varphi$ .*

For example,  $\varphi(x, y) = \neg x \wedge y$  is a satisfiable instance of the SAT-problem, namely for  $x = 0, y = 1$ . Recall that it has been shown that the general SAT-problem is NP-complete, meaning that solvers will need to rely on efficient heuristics to provide adequate performance. Applying solvers for the SAT-problem to real-world problems additionally relies on transformations from the given problem semantics to a boolean formula. This process can be involved. Instead, one can use solvers that incorporate more expressive theories.

**Definition 2. SMT-problem.** *Given a first-order logic formula  $\varphi$  over a set of theories  $T = \bigcup_{i=0}^m T_i$ , decide whether  $T \models \varphi$ , i.e.  $\varphi$  is satisfiable in  $T$ .*

The SMT-problem poses the question whether an assignment of the formula’s variables to some values exist such that  $\phi$  can be satisfied. Here,  $\phi$  can be composed of elements of multiple theories, such as the theory of integer arithmetic, arrays or reals, and is not restricted to the boolean domain. As an example, one can ask an SMT-solver to check

for the satisfiability of  $\varphi(x, y, a) = x > 0 \wedge y = a[x] \wedge y = \pi$ , which is obviously satisfiable, e.g. for  $x = 1$ ,  $a = [0, \pi]$ ,  $y = \pi$ .

In model checking, solvers are widely applied to encode transition systems and property violation checks. The concrete transformation of the system to a logical formula will obviously vary between techniques. Once such an encoding is constructed, it is passed on to the solver which then either refutes the formula or provides a satisfiability witness. In the first case, one has found no property violation, where the latter case provides a counterexample.

### 3.2.2 Bounded Model Checking

Bounded model checking is a typical under-approximation technique: The possibly infinite program state space is unrolled up to a given bound in a breadth-first manner [Biere et al. 1999a]. The bound is fixed before the verification process starts. The finite sub-space is then checked against property violation. Hence, only finitely many program paths with an  $\omega$ -regular representation are examined.

For a given bound  $k$ , a bounded model checking instance  $BMC_k$  can be encoded by means of formulae that are passed to a satisfiability solver:

$$BMC_k(s_0, \dots, s_k) = I(s_0) \wedge \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left( \bigvee_{i=0}^k \neg P(s_i) \right) \quad (1)$$

Formula 1 starts with all initial states  $s_0$  and applies the transition relation  $T$  iteratively until the bound  $k$  is reached. Then,  $\neg P$  checks whether any state that does not fulfill the property was discovered. If so, the formula becomes satisfiable and the solver produces a counterexample witness.

This technique finds shallow bugs with ease, but fails to locate counterexamples hidden deep inside the execution of the program. This can be cushioned by using a goal-directed search technique, moving away from the classic breadth-first expansion. In practice, bounded model checking can often not fully expand extremely large state spaces, thus not deliver proofs on those, which is of particular importance in the case of embedded-style programs with their unbounded main loop. Figure 7 depicts two C programs. The first contains a shallow bug at depth 10 that is reachable if the unroll depth is set accordingly. The second program contains a bug once  $x$  overflows and reaches 0 again. This happens in the very last unrolling of the main loop, thus the complete state space of  $2^{32} \cdot 2^{31} = 9\,223\,372\,036\,854\,775\,808$  elements had to be examined to find the bug. In practice, unrolling a loop in the realm of quintillions can be quite challenging.

```

1 extern void __VERIFIER_error();
2 int main() {
3     int x, y = 0;
4     while (1) {
5         x++;
6         while(y < x) y++;
7         y = 0;
8         if (x == 10) {
9             __VERIFIER_error();
10        }
11    }
12 }

```

```

1 extern void __VERIFIER_error();
2 int main() {
3     int x, y = 0;
4     while (1) {
5         x++;
6         while(y < x) y++;
7         y = 0;
8         if (x == y) {
9             __VERIFIER_error();
10        }
11    }
12 }

```

Figure 7: Two sample embedded-style C programs, where the left is easy to solve for bounded model checking, and the right is unprovable unless the state space was to be completely unrolled.

### 3.2.3 Incremental Bounded Model Checking

Instead of providing a fixed bound at the beginning of the verification process, one can also incrementally increase the unwind depth and check against a potential property violation in the newly discovered states of each increment [Biere et al. 1999b]. The verification process terminates if this check was successful and increases the current unwind depth otherwise.

Bugs that lie shallow are found very early during an incremental strategy, whereas standard bounded model checking with a high-depth configuration will need more computation time for the additional unwindings. One could tackle that by choosing a lower bound, but in exchange of missing out on deeper bugs. Hence, incremental bounded model checking is ideal for unbounded loops as present in embedded-style programs. Although potentially improving the performance of bounded model checking, an incremental technique does not lead to an increased bug-coverage.

### 3.2.4 k-Induction

$k$ -induction is a generalization of the well-known principle of mathematical induction. With plain induction, a proof of  $\forall n \in \mathbb{N}. A_n$  is delivered by showing that  $\forall n \in \mathbb{N}. (A_n \implies A_{n+1})$  holds. Furthermore, the base case  $A_0$  has to be proven. While induction is powerful on its own, it can not be applied to universally proof arbitrary hypothesis.

We can strengthen it by improving on the induction hypothesis – instead of just  $A_n$ , we now assume for a fixed  $k \in \mathbb{N}^+$  that  $A_{n-k+1} \wedge \dots \wedge A_n$  holds. This leads to the principle of  $k$ -induction: For some  $k \in \mathbb{N}^+$ , we show that  $\forall n \in \mathbb{N}. (\bigwedge_{i=n-k+1}^n A_i \implies A_{n+1})$  as well as  $\bigwedge_{i=0}^{k-1} A_i$ .

In the context of program verification,  $k$ -induction can be used to verify loops [de Moura et al. 2003].  $A_n$  then states the correctness of iteration number  $n$ . This allows the verifier to look back a fixed amount of arbitrary execution steps in the loop and derive correctness from there. As the look-back increases, so does the load on the prover due to assumption accumulation.

Again, we can write the induction step of a  $k$ -induction task using a formula  $IND_k$  over the program states:

$$IND_k(s_0, \dots, s_k) = \left( \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left( \bigwedge_{i=0}^{k-1} P(s_i) \right) \wedge \neg P(s_k) \quad (2)$$

In comparison to bounded model checking, equation 2 misses the initial condition  $I$ . Further, only states in which the property  $P$  holds are selected for the induction hypothesis. Via  $\neg P$  it is then checked whether the property holds under the induction hypothesis. Since this formula can be passed to a satisfiability solver,  $k$ -induction can be integrated into a bounded model checker as they are already equipped to build such formulae.

Additionally, if one does not want to change the inner workings of a verifier, one can easily implement  $k$ -induction on top of a bounded model checker by applying code transformations to the program [Gadelha et al. 2017]:

- **Base case:** Just run standard bounded model checking on the program.
- **Forward case:** Check if the base case has unrolled the program completely.
- **Induction case:** Transform the input program such that all input variables are set to a non-deterministic value once the main loop is entered. We then assume the property on the beginning of each main loop iteration and check it after the verifier has unrolled  $k$  iterations.

Here, we assume that an embedded-style program is given as input. If the base case finds a bug, we terminate with a *False* answer, if the induction case does not find a bug for a given  $k$  and the base case has already been unrolled up to  $k$  steps, we terminate with a *True* answer. If the forward case indicates that the program was unrolled completely,

we return the answer of the base case. If neither is the case,  $k$  is increased and base and induction case are run again.

<pre> 1 extern void __VERIFIER_error(); 2 3 int main() { 4   initialize(); 5 6 7   while(1) { 8 9 10    step(); 11    if(!property()) 12      __VERIFIER_error(); 13  } 14 }</pre>	$\rightsquigarrow$	<pre> 1 extern void __VERIFIER_error(); 2 extern void __VERIFIER_assume(int); 3 int main() { 4   initialize(); 5   set_loop_variables_nondet(); 6   unsigned int i = 0; 7   while(1) { 8     __VERIFIER_assume(property()); 9     i++; 10    step(); 11    if(i == k &amp;&amp; !property()) 12      __VERIFIER_error(); 13  } 14 }</pre>
--	--------------------	---

Figure 8: The transformation applied in the  $k$ -th induction step.

Figure 8 shows the described transformation of the induction case in detail. The function `set_loop_variables_nondet()` sets each variable that is *modified* in the loop to a non-deterministic value of its associated type. Then, before entering each loop iteration, the induction hypothesis is emulated by the statement `__VERIFIER_assume(property())`.

Although theoretically necessary, the forward case does not play a role for our case studies as their main loops are unbounded. Hence, the forward case will be ignored in the upcoming discussion. To improve performance, both processes can be run in parallel. Additionally, the bounded model checker can be replaced by an incremental bounded model checker such that loop unrollings are not repeated for every new  $k$ . In that case, we can remove the additional variable `i`, as incremental bounded model checkers already check only the last unrolling for property violations.

An illustration of the iterative process of  $k$ -induction is presented in Figure 9. In the example, a program  $C$ , which does not contain any property violations, is proven correct for  $k = 3$ .

Firstly, all loop variables are initialized to a non-deterministic value. This encompasses the complete state space of  $P \cup \bar{P}$ , where  $P$  represents all states satisfying the property. Then, the first induction hypothesis assumes that the property is valid. Thus, for  $k = 1$ , the hypothesis is  $P$ . Once the hypothesis is assumed, the `step()` function is executed. Specifically, the step semantics is applied to each hypothesis state. This application refines the set of reachable states, as we started with the general assumption that all states are reachable, and now discover step by step which states are actually valid under the loop’s semantics. On the other hand, this process may also lead to the discovery of property violations. Such a violation is depicted for  $k = 1$ , implying that the property could not be proven – analogous to spurious counterexamples of other techniques. To

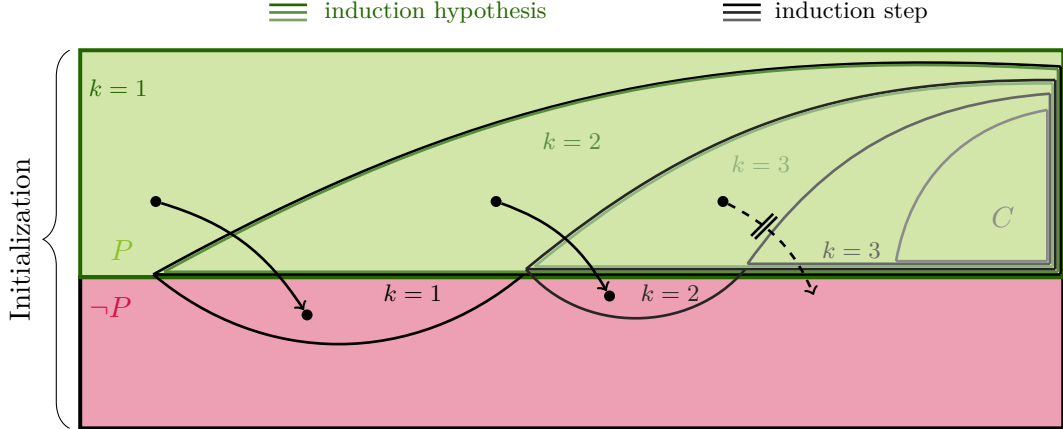


Figure 9: An illustration of the iterative refinement of the state space during  $k$ -induction. Here, the program  $C$  is valid, i.e. does not violate the property  $P$ .

refine the verification, we set  $k = 2$ . Now, the hypothesis is constructed by taking all previously reached states and prune the invalid ones. Namely, the hypothesis for  $k = 2$  contains all states of the induction step for  $k = 1$  that are contained in  $P$ . The previously described steps are repeated until a refinement was found for which no more counterexamples are reachable. In the example, this holds for  $k = 3$ . When applying the semantics of the step function to the induction hypothesis of  $k = 3$ , no property violating state can be found. Thus,  $k$ -induction has identified a representation that encloses the loop semantics fully but does not contain any property violation.

The illustrated process depicts the strength of  $k$ -induction: it explores the inherent structure of the main loop, such as the property being checked after each iteration and each step being bounded. Thus, it was shown to be well-applicable to embedded-style programs [Rocha et al. 2016]. As  $k$ -induction over-approximates the loop semantics and iteratively refines the set of reachable states, it can be classified as an over-approximation technique.

The described process does not allow a direct application to multi-loop programs. Each loop step has to be bounded – otherwise, the verifier will not provide an answer even for  $k = 1$ . There exist several techniques that make  $k$ -induction applicable to multi-loop programs [Donaldson et al. 2011], but those are not considered in this thesis as they are not in the scope of our case study.

While  $k$ -induction can deliver promising results, it inheres the conceptual flaw of the look-back not being able to distinguish between a reachable and unreachable induction hypothesis. In the latter case, we may not be able to infer a proof, although the property is actually valid. Figure 10 shows such a case, following the example of [Beyer et al. 2015]. Here,  $k$ -induction assumes a non-deterministic integer for each variable  $x$ ,  $y$ , and  $z$ , although  $z$  is fixed to be in  $\{0, 1\}$ . This leads to the counterexample starting with

```

1 extern void __VERIFIER_error();
2 int main() {
3     int x = 0, y = 0, z = 0;
4     while (1) {
5         if (z == 0) x++;
6         if (z == 1) y++;
7         if (z < 2) z++;
8         if (z == 2) z = 0;
9         if (z == 0 && x != y) __VERIFIER_error();
10    }
11 }

```

Figure 10: A program for which plain  $k$ -induction can not deliver a proof.

$z = -k$ ,  $x = 0$ ,  $y = 1$  for every induction step except when  $k$  is equal to the integer domain cardinality. In that case, the hypothesis starts from the beginning of the program execution and  $k$ -induction can distinguish between reachable and unreachable program parts. It should be noted that setting  $k$  to such a high value is often impractical, and one could replace  $k$ -induction with bounded model checking of the highest possible bound anyways as this would also verify the program.

This problem can be ascribed to a *weak induction hypothesis*. Respectively, it can be solved by introducing additional propositions to the induction hypothesis in order to strengthen the proof and avoid starting from unreachable loop prefixes [Beyer et al. 2015]. This can be done in a multitude of ways. For example, the developer can annotate additional loop invariants, or variable value domains can be inferred by static analysis. In the exemplary program, an addition of `__VERIFIER_assume(z == 0 || z == 1)` as the first loop body's statement allows  $k$ -induction to verify the program with ease.

### 3.2.5 Symbolic Execution

To avoid storing huge transition systems in the computer's memory while exploring program state spaces, one can employ symbolic techniques [McMillan 1993]. This is vital for non-deterministic assignments such as `int x = __VERIFIER_nondet_int();`. In a plain explicit setting, every possible integer value would be stored, leading to a state space with a cardinality of 4 294 967 950 for just one 32-bit integer variable. Turning away from such an explicit storage typically trades decreased memory usage for increased computation load.

In a nutshell, variable values are extended from being a single element of their domain to having a symbolic value. Where an explicit value is a basic equality, e.g. `x = 5`, symbolic value representation poses constraints over the variable value domain, e.g. `0 < x && x < 10`. If we reduce those constraints to one assignment, we reach the special case of explicit value storage. Constraints can now be collected in the format of a formula



rather shifts load from storage to processing. Considering our example, it means that the invalidity of “if ( $z < x$ )” can not be directly inferred as the explicit values are not given. We rather need to check the satisfiability of the inequality considering the existing constraints collected along the paths. This can be done by employing a satisfiability solver, which in turn increases computation time.

### 3.2.6 Explicit Analysis

An explicit analysis keeps track of every variable value during program execution. It is a straightforward approach to the storage of programs states [Beyer and Löwe 2013]. Plain explicit analysis builds the complete transition graph of a program by tracking the value for each variable. It then runs property verification algorithms on this graph which itself is stored in an *explicit* manner – as opposed to symbolic – meaning that every variable value change in the program is directly reflected and present in the transition system. An explicit value analysis on its own is similar to bounded model checking with the bound set to infinity.

As programs can become extremely large – although not infinite – in practice, explicit analysis is often enhanced with the techniques discussed in the following. It can therefore act as a canvas to start building a more efficient verification process. For example, abstraction techniques can be employed where only certain value domains of the variables are tracked.

### 3.2.7 Abstractions and CEGAR

A way of reducing potentially enormous state spaces without under-approximation is *abstraction*. The approach is based on not relying on the exact semantics of the given program, but rather using a coarser semantics to approximate the actual program [Clarke et al. 1994]. This leads to abstract states representing multiple concrete states, as demonstrated in Figure 12.

Here, the actual state space is visible in blue and its over-approximation gray. The abstraction may also include states that are not present in the actual program semantics, as indicated by upper-most abstract state that is not intersecting with the exact state space.

In case a verifier decides to employ an over-approximating abstraction technique and it detects a proof, it is safe to assume that the proof also holds in the original program. On the other hand, if a counterexample is found, a definite assertion about the property’s invalidity under the original semantics can not be made directly. Only a simulation under exact semantics can reveal its validity in the original program. Such a counterexample can be seen in Figure 12, drawn in red. It crosses the original semantics, although it is not completely contained in the exact state space. Hence, a verifier based on abstraction would need to replay such a counterexample with a non-abstracting technique such as

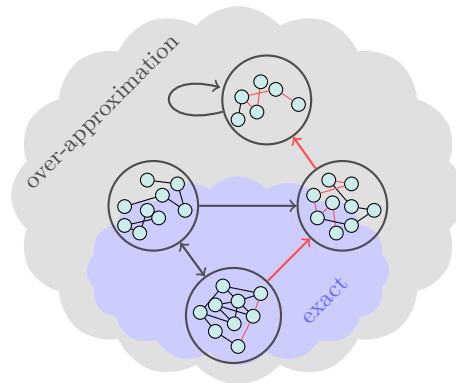


Figure 12: An illustration of an exact program state space and an over-approximating superset. Multiple concrete states are contained in four abstract states as a result of abstraction. A spurious counterexample is depicted in red.

bounded model checking. In the exemplary case, it will find that the counterexample is actually invalid under the original program semantics, as the replayed path of the concrete states moves outside the exact state space. Then, another – hopefully valid – counterexample has to be searched for.

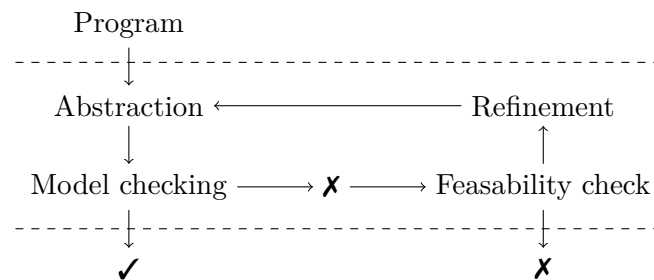


Figure 13: The CEGAR procedure. The program abstraction is iteratively refined until either a proof or a feasible counterexample is generated.

With any abstraction, the goal is to find a resolution that is sufficiently coarse for an efficient analysis, but also just fine enough to not include spurious counterexamples – this is the leverage-point of *Counter-Example Guided Abstraction Refinement (CEGAR)* [Clarke 2003]. Figure 13 depicts its schematic process. Here, we start with a very coarse abstraction that is constructed from the input program. Model checking is then run on the abstraction. If an undesirable spurious counterexample is found as a result, CEGAR uses the information from that invalid program path to refine the abstraction such that the infeasible counterexample is not found again. This process is repeated iteratively until either a proof or a feasible counterexample is found.

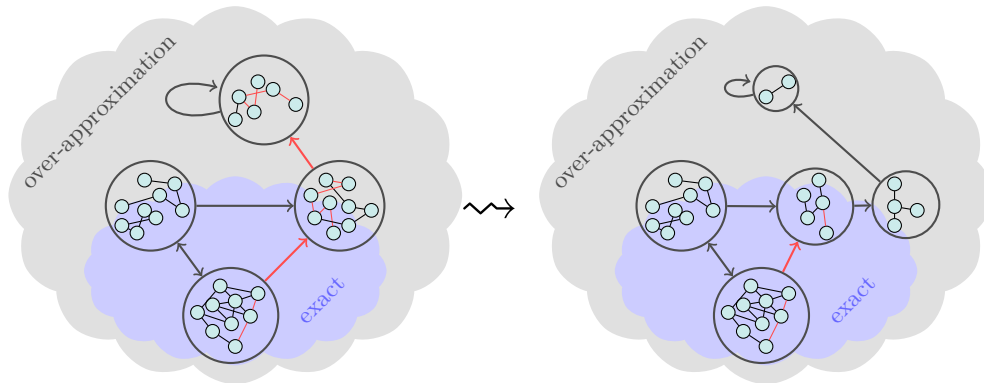


Figure 14: An overview of one CEGAR iteration. The spurious counterexample on the left is identified using an exact technique and excluded from the new abstraction on the right.

A graphical outline of one CEGAR iteration based on the previous example can be found in Figure 14. The spurious counterexample is identified by replay, and excluded accordingly. This leads to a refinement of the right- and uppermost abstract states, which in turn allows the analysis to find a more precise counterexample that is contained in the exact program semantics.

CEGAR is often combined with *lazy abstraction*, meaning that the refinement only takes place along those abstract states that are actually included in the spurious counterexample [Henzinger et al. 2003]. Instead of refining each abstract state in the current transition system, the refinement is then contained to fewer states while the goal of excluding the counterexample is still fulfilled.

### 3.2.8 Predicate Abstraction

The previous section described the general application of abstraction, but left open how such an abstract state space can actually be constructed. There are several methods of characterizing abstract states, one of which is *predicate analysis* and presented here exemplary [Graf and Saidi 1997]. As the name suggests, it stores the abstract states as a set of predicates in a quantifier-free theory over the program variables. For example, given the variables  $x$  and  $y$  in the integer theory, one can store predicates such as  $x \neq 1$  or  $x \geq y$ .

The concrete program states are assigned to the abstract ones depending on their evaluation of the specified predicates. The abstract semantics then contains a boolean variable for each predicate that is tracked. To then check for error reachability, one needs to check the value of the predicate containing the property. Iteratively refining the set of tracked predicates allows the exploitation of a CEGAR procedure.

The process of integrating any abstraction technique into CEGAR requires two steps:

1. *Interpolating* facts about the infeasible counterexample such that it is excluded.
2. *Refining* the abstraction based on the interpolated facts.

**Interpolation:** Given two formulae  $A$  and  $B$  s.t.  $A \wedge B$  is unsatisfiable, a formula  $I$  is an interpolant if  $I \wedge B$  is unsatisfiable,  $A \implies I$ , and  $I$  only uses symbols of both  $A$  and  $B$ . Such interpolants can be calculated by SMT-solvers using various techniques, e.g. Craig interpolation [McMillan 2005]. It has been shown that an interpolant always exists for two contradicting first-order formulae  $A$  and  $B$  [Craig 1957]. Intuitively, an interpolant removes information from  $A$  that the solver gathered by showing that  $A$  and  $B$  are inconsistent. The property in question, i.e. the inconsistency of  $A$  and  $B$ , still holds for the interpolant, but it is allowed to contain less information as  $A$  since  $A \implies I$  holds.

In the context of a spurious counterexample, interpolants are computed along the transitions of its path. Analogue to the previously introduced bounded model checking approach, paths can be represented symbolically as formulae by means of state formulae over the set of states  $S$  and transition formulae over  $S \times S$ . Having identified a spurious counterexample, for each state  $k$  leading up to the error state, the path is divided into the part prior to  $k$ ,  $A_k$ , and the part after  $k$ ,  $B_k$ . Hence, the path  $A_k \wedge B_k$  is inconsistent under the concrete semantics, as are its interpolants  $I_k$  with  $B_k$ . However, as  $A_k \implies I_k$ , the interpolant contains information that is valid along the path prior to  $k$ . Once all interpolants are computed, we have gathered enough information to exclude the counterexample in upcoming iterations, as each interpolant is inconsistent with a part of the path leading to the error location.

**Refinement:** A refinement strategy takes the spurious counterexample and the interpolants that were computed along its path and refines the current abstraction level accordingly. Most commonly, two strategies are observed in today's verifiers: *Interpolant refinement* [McMillan 2006] and *predicate refinement* [Jhala and McMillan 2006].

The first case does not rely on a predicate abstraction. Instead, interpolants are directly used to refine the states. Recall that interpolants are available for each abstract state  $k$  along the error path. Hence, a simple conjunction of the interpolant  $I_k$  and the current formula representing the abstract state  $k$  suffices to make the counterexample unreachable from  $k$ .

On the contrary, the latter case updates the predicates of each abstract state with the information from the interpolants. As predicates have to be quantifier-free, the atoms of the interpolants are added to the predicates of the states along the abstract counterexample.

### 3.2.9 Trace Abstraction

Predicate analysis removes infeasible counterexamples from its abstraction until the verifier reaches a definite answer. A trace abstraction on the other hand *adds* infeasible counterexamples traces to its abstraction until it can either show that all possible error traces of the program are covered by those infeasible counterexamples, or a non-feasible counterexample was found [Heizmann et al. 2009]. Hence, instead of over-approximating the program semantics, we now under-approximate its inverse behavior. Again, a CEGAR approach is employed as the abstraction is iteratively refined by the information from the infeasible counterexamples, which can be stored using automata. Those argue over the set of program traces, i.e. possible executions identified by a sequence of statements. For example,  $x := 0, x := x + 1$  can be such a trace.

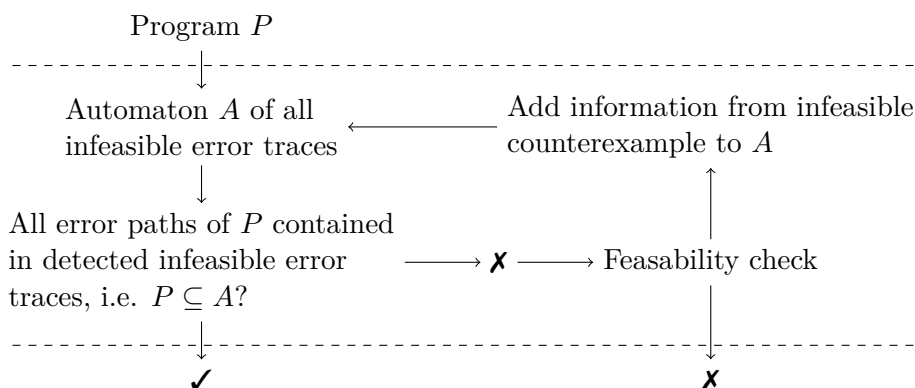


Figure 15: An automaton-based CEGAR procedure.

Figure 15 gives an illustration of the integration into a CEGAR procedure. The abstraction is represented as an automaton whose language is a set of infeasible counterexamples. It is then checked if the program contains error traces – whether actually reachable or not – that are not yet identified as infeasible, i.e.  $P \not\subseteq A$ . If no such path can be found, the program is found valid. Otherwise, we have identified an error path that has to be checked for feasibility. If the path is valid under the concrete semantics, we return. Otherwise, the automaton  $A$  is refined with the information from that counterexample. The most straightforward method implements a simple addition of the path to  $A$ , but this would lead to little information added in each iteration. As previously discussed, interpolants deliver a way of extracting more information from infeasible counterexamples, specifically about the reason of their infeasibility. In this abstraction, interpolants are computed for each statement in the trace, leading to a sequence of predicates.

Amongst various techniques, *unsatisfiable cores* can be employed for interpolation [Heizmann et al. 2014]. For an unsatisfiable formula, a solver can identify the refuted subformulae that lead to the formula’s unsatisfiability. The set of such formulae is called the unsatisfiable core. In the construction of the automaton representing the infeasible error

traces, the unsatisfiable core ultimately represents the statements that lead to reaching the error location. Hence, one can ignore other statements and compute the sequence of predicates along the remainder of the trace.

### 3.3 C software verifiers

The efficiency of the previously presented verification approaches has been thoroughly examined by means of implementations by various research groups. Today, we observe a wide range of matured academic verifiers providing a high performance while keeping the required operational expertise to a minimum. In order to analyze the performance of those verifiers on our specific use case of embedded automotive C code from Simulink models, we selected a suitable subset of verifiers based on the following criteria:

1. Supports the previously discussed features of the case studies, e.g. precise operations on floats and bit vectors.
2. Allows for an in-code specification of the property.
3. Has matured enough to compete in the SV-COMP, shows promising results on the relevant tasks and competes in the *Overall* category.
4. Has to be actively maintained in the last two years.

The rationale behind those requirements arises as following:

1. If the verifier delivers a result, it has to be precise. It shall not return false positives or negatives.
2. It is not desirable to support various property specification formats for different verifiers. In a streamlined verification process, the property can be specified in such a way that every verifier is able to operate on. This is ensured by an in-code specification, especially as the properties can be checked in that manner due to their validity check once per main loop iteration. Conveniently, every verifier adhering to the SV-COMP standards supports the usage of `__VERIFIER_error()`.
3. Newly developed verifiers are prone to bugs and may deliver false positives. Such bugs can be hard to identify, particularly if the faulty verifier is the only one to deliver a result on that specification. Verifiers competing in the SV-COMP with a respectable score have proven their maturity. High scores on tasks relevant to ours, such as *ReachSafety*, can be a primer of good performance. As our case studies are quite involved, a participation in the *Overall* category again serves as an indicator of a well-engineered verifier.
4. Verifiers that are not somewhat actively maintained are likely to contain bugs, hence the reasoning of 2. applies.

To be able to compare the proficiency of the verifiers with regards to the features of our

case study, we extracted a custom score from the SV-COMP 2018 results<sup>2</sup> representing only relevant subcategories. Those are: *ReachSafety-Arrays*, *ReachSafety-BitVectors*, *ReachSafety-ControlFlow*, *ReachSafety-Floats*, *ReachSafety-Heap*, *ReachSafety-Loops*, and *SoftwareSystems-DeviceDriversLinux64\_ReachSafety*. Although not completely applicable – e.g. *ReachSafety* contains a lot of small, rather academic examples, and *SoftwareSystems* is not based on embedded systems – it should still give a primer on what to expect from the verifiers.

The score is calculated as follows, where  $a_c^v$  is the score for verifier  $v \in V$  in category  $c \in C$  and  $\min c$ ,  $\max c$  the minimum resp. maximum points in a given category: First, the relative placing  $r_c^v$  of the verifier in a given category is calculated via  $r_c^v = \frac{a_c^v - \min c}{\max c - \min c}$ . This places all verifiers on a scale from 0 to 1 depending on how close they got to a full maximum score. The verifier with the lowest score represents zero. From this relative score for each category, one can now assemble a weighted joined score  $r^v$  for all categories. This is done by choosing a weight  $w_c$  for each category such that  $\sum_{c \in C} w_c = 1$  which is then used to calculate  $r^v = \sum_{c \in C} w_c \cdot r_c^v$ . This joined relative score again places the verifiers on a scale from 0 to 1 depending on their relative overall result.

Table 4: The category weights used in the relative score calculation.

<i>Category</i>	<i>Weight</i>
ReachSafety-Arrays	0.05
ReachSafety-BitVectors	0.25
ReachSafety-ControlFlow	0.25
ReachSafety-Floats	0.25
ReachSafety-Heap	0.05
ReachSafety-Loops	0.05
SoftwareSystems-DeviceDriversLinux64_ReachSafety	0.10

The weights are fit to the specifics of our case study and are given in Table 4. We decided to focus on good results on bit vectors and floats as well as control flows structures since all of those features are heavily present in the embedded code.

The final scores are presented in Table 5, where only verifiers have been analyzed that competed in the *Overall* category. CPACHECKER has accumulated the highest score amongst the selected subcategories, directly followed by ULTIMATEAUTOMIZER and ESBMC-K-INDUCTION. The rest of the verifiers fall pretty closely in range between 0.81 and 0.75, except for INTERPCHECKER. Thus, we included CPACHECKER, ULTIMATEAUTOMIZER and ESBMC, but did not include the other tools from the ULTIMATE tool chain, as its best contestant, ULTIMATEAUTOMIZER, is already present and they have shown comparable results in the SV-COMP 2018. Besides that, the goal was to select a subset of tools that represent a variety of verification techniques. Thus, the bounded model checker CBMC was included, as it forms the basis of some of the other

<sup>2</sup>Results were taken from <https://sv-comp.sosy-lab.org/2018/results/results-verified>

Table 5: The final relative scores and their placings for each verifier. Selected verifiers are marked in bold.

<i>Place</i>	<i>Verifier</i>	<i>Relative score</i>
1	<b>CPAChecker</b>	0.872
2	<b>UltimateAutomizer</b>	0.856
3	<b>Esbmc-k-induction</b>	0.828
4	UltimateTaipan	0.806
5	Symbiotic	0.792
6	<b>2LS</b>	0.772
7	UltimateKojak	0.771
8	<b>CBMC</b>	0.770
9	<b>ESBMC-incremental</b>	0.757
10	DepthK	0.756
11	InterpChecker	0.206

tools presented. Although not competing in the recent SV-COMP, SMACK delivered very promising results in the 2017 version, especially on the *ReachSafety* and *SoftwareSystems* categories, in both of which it won a gold medal. As an exception, we therefore included it in our selection. Finally, 2LS was chosen to represent a more sophisticated  $k$ -induction approach, as such techniques have shown promising results on embedded code. Lastly, we also benchmarked the case studies on a custom-developed  $k$ -induction tool that works on top of any bounded model checker.

Figure 16 provides a view on the internal state space representations constructed by the verifiers. While some tools – e.g. CBMC – rely solely on an under-representation, other verifiers use a wide spectrum of techniques – e.g. CPACHECKER and 2LS. Most approaches are of an iterative nature, refining the state space representation to be more exact as the verification continues. Note that this only surveys the inner workings and has no implications on the validity of the final result of each verifier, which is always precise. We present conceptual approaches of each verifier with regards to the previously described techniques as well as their strengths and weaknesses in the upcoming sections.

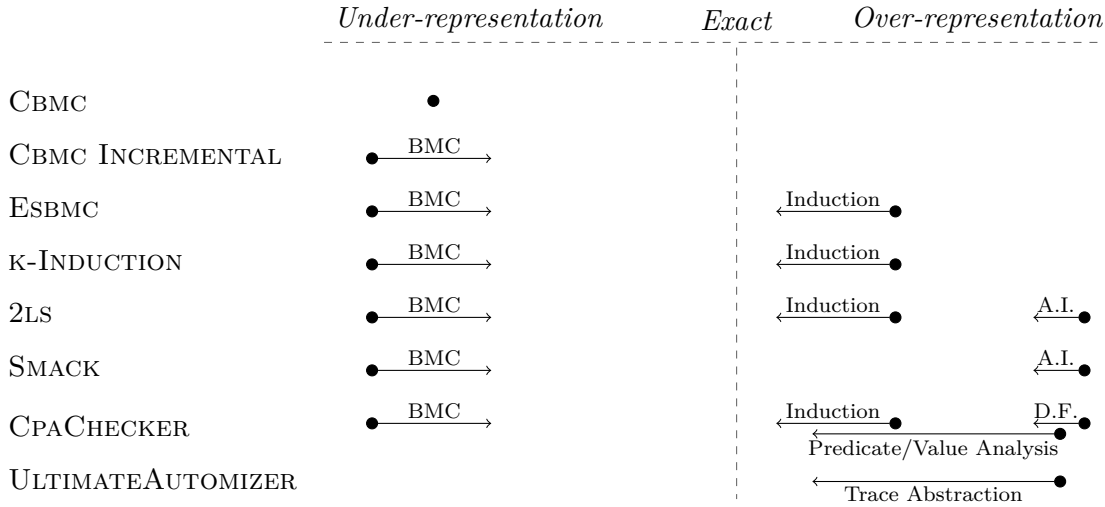


Figure 16: An overview of the internal state space representations for the examined configurations, including the approximation direction towards the exact space. Abbreviations: *A.I.* = Abstract Interpretation, *D.F.* = Data Flow Analysis.

### 3.3.1 CBMC

The *C Bounded Model Checker* (CBMC) is a bounded model checker for C programs and checks, among others, against user-specified assertions, overflows, and pointer safety [Clarke et al. 2004]. It unwinds each loop in the program up to a bound specified by the user. This bound can be set on a per-loop basis or generally for all loops. To tackle the state space explosion problem, it employs symbolic execution during program exploration. Firstly presented in 2004 by Edmund Clarke et al. at the Systems Verification Group of the University of Oxford, it has been actively maintained up to this day.

Internally, C programs are first converted to a GOTO-normal form in which all loops and conditional branches are transformed to guarded GOTO-statements. CBMC then performs static analysis to resolve function pointers into a static set of candidates. This normalized program with a static call graph is converted into a static single assignment (SSA) form, a normal form where each variable is assigned to exactly once. This is done so that the formula contains different variables for each loop iteration. The unwinding process then produces a formula over the boolean theory in conjunctive normal (CNF) form which is passed to an SAT-solver. If a satisfiable assignment was found by the solver, CBMC extracts a property-violating program trace and returns **VERIFICATION FAILED**. Otherwise, it prints **VERIFICATION SUCCESSFUL** – although the program may not be fully unwound.

In a default configuration, CBMC reduces all of the program semantics – e.g. bit-vectors, arrays and floating point arithmetic – to boolean satisfiability problem by means of

a bit-precise internal translation. The tool also supports various SMT-solvers where semantics of supported theories can be directly modeled instead of being internally translated [Cordeiro et al. 2012].

**Strengths:** CBMC has matured since 2004 and has proven to be extremely efficient for finding (shallow) bugs, placing second, first and fourth places in the *Falsification Overall* category in the SV-COMP 2018, 2017, and 2016 respectively. CBMC’s success has lead to a variety of forks building additional verification techniques upon its simple but effective approach to bounded model checking.

**Weaknesses:** Besides unrolling a program completely, CBMC implements no additional proof techniques such as  $k$ -induction or any abstraction concepts. Hence, for the case of unbounded embedded programs, CBMC can only be employed for bug-hunting, as the state spaces of the case studies are impractically large. Additionally, CBMC has to be given a bound prior to starting the verification process and is not able to increase a bound dynamically in case it has not detected a counterexample but still left paths unexplored. The SV-COMP version of CBMC solves this problem by using a wrapper script that sets bounds of 2, 6, 12, 17, 21, and 40 as long as neither a counterexample was found nor the time limit was exceeded. Those numbers were taken from practical observations [Kroening and Tautschnig 2014].

### 3.3.2 CBMC Incremental

CBMC INCREMENTAL is a more elaborate approach to the fixed a-priori bound of CBMC [Schrammel et al. 2015]. It is a fork of the 2016 version of CBMC adding additional incremental support which can be either enabled for all loops, or in a per-loop setting where the rest of the loops are unwound either fully or up to a fixed bound. After its initial fork by Peter Schrammel in 2014 it has only been maintained for two years, but has been the only attempt to integrate a plain incremental setting into CBMC.

**Strengths:** Although currently not maintained, the tool’s applicability to unbounded embedded-style programs such as our case study gives great advantages. This means that no loop bound has to be guessed beforehand, as the main loop can just be unwound incrementally until a timer expires. In contrast to the wrapper script of CBMC, the previously constructed state space is not torn down and re-constructed after each increment. Furthermore, only the last unwinding is checked – as all prior states have been proven correct in earlier unwindings. This lessens the burden on the solver.

**Weaknesses:** Apart from introducing dynamically increasing bounds, CBMC INCREMENTAL does not add any proof techniques. Thus, the weaknesses that apply to CBMC also hold here.

### 3.3.3 ESBMC

The *Efficient SMT-based Bounded Model Checker* (ESBMC) was forked off of a 2008 version of CBMC and has been replacing original framework parts ever since [Gadelha et al. 2018]. As of today, most of CBMC’s core has been rewritten. The main driver behind creating a fork was updating CBMC to directly translate to SMT-theories instead of relying on SAT-solvers. Additionally, ESBMC implements an efficient concurrency model using partial order reduction.

Of special interest to our case study is ESBMC’s incorporated incremental bounded model checking and  $k$ -induction support. This enables ESBMC to provide proofs and makes fixed bounds superfluous.

**Strengths:** ESBMC with its  $k$ -induction and incremental mode is specifically applicable to embedded code while the general concept of CBMC – a bounded, symbolic unwinding of loops – still remains. Performance improvements can be expected from using SMT-solvers as one can encode certain theories directly. This gives the SMT-solver more details of the program’s semantics which in turn can lead to a more efficient proof generation.

**Weaknesses:** ESBMC provides a very basic version of  $k$ -induction where no additional invariants are deduced to aid the induction process. As presented in section 3.2.4, some programs can not be proven with plain  $k$ -induction unless the complete state space is unrolled. Further, and in contrast to CBMC, one can not fine-tune loop bounds with ESBMC – only one bound can be specified for all loops present in the program.

### 3.3.4 2LS

2LS (read: *tools*) is another fork of CBMC, expanding bounded model checking to a multitude of verification approaches [Schrammel and Kroening 2016]. It interprets program analysis as a problem of solving a second order logic instance and reduces this to quantifier elimination in first order logic. This leads to a variety of concepts that 2LS can employ, as long as their approaches can be modeled using second order logic. Thus, 2LS can be seen as a framework for verification techniques. Among others, this framework includes (incremental) bounded model checking,  $k$ -induction,  $k$ -induction  $k$ -invariants, and abstract interpretation over various domains such as intervals and octagons [Cousot and Cousot 1977].

Initially, a verification task is specified by a second order logic formula. Similar to bounded model checking in SAT, one can verify safety properties by checking

$$\begin{aligned}
 SAF = \exists_2 Inv. \forall s. \forall s'. (I(s) \implies Inv(s')) \wedge \\
 ((Inv(s) \wedge T(s, s')) \implies Inv(s')) \wedge \\
 (Inv(s) \implies \neg P(s))
 \end{aligned} \tag{3}$$

where  $\exists_2$  is a quantifier in second-order logic. Similarly, one could specify conditions for termination or invariant derivation.

As second order logic itself is undecidable, and solvers for decidable fragments thereof are still scarce [David et al. 2014], one has to find a first-order logic formula for *SAF* by removing  $\exists_2$  which in turn can be passed to a first order logic solver. This transformation applies to each invariant and replaces them with templates, a technique originally used in program synthesis. A template represents a fixed expression over the program state and its template parameters. The task of the first order logic solver is then to find such parameters that fit the invariant.

Besides plain  $k$ -induction, 2LS also supports an extension called  $k$ -induction  $k$ -invariant. Here, in each iteration the induction hypothesis is strengthened by a  $k$ -invariant. This invariant is found by abstract interpretation techniques and depends on the information learned in previous  $k - 1$  iterations. Specifically, it contains an over-approximation of the information that each path has to start in an initial state, the appearance of the encountered states, and that the states encountered in  $k - 1$  unwindings are safe. Thus, abstract interpretation is strengthened by the information learned in the  $k$ -induction process, and  $k$ -induction in turn is strengthened by the invariant extracted from abstract interpretation. Moreover, even the bounded model checking part of  $k$ -induction profits from a stronger invariant, as this prunes parts in the symbolic execution. The authors of 2LS call this observation a “a genuine synergy between components” [Brain et al. 2015].

As its back-end, 2LS uses SAT-solvers due their better support of the required incremental solving. Modeling the C semantics as well as creating the boolean formula is covered by the CBMC core.

**Strengths:** As 2LS combines  $k$ -induction with over-approximating static analysis to deduce stronger invariants, potentially more properties can be verified compared to plain  $k$ -induction which in turn could lead to an even better applicability to our case study.

**Weaknesses:** 2LS is limited if a so called *irreducible control flow* is present in the control flow graph. This is the case if a loop has more than one entry point, e.g. through `goto` statements. Besides, its invariant generation has potential for improvement when arguing about arrays and pointers [Schrammel and Kroening 2016].

### 3.3.5 SMACK

Rather than being a verifier by itself, SMACK, a project Microsoft Research initiated in 2013, translates from the LLVM intermediate representation (IR) into the BOOGIE intermediate verification language (IVL) [Rakamarić and Emmi 2014]. This enables the verification of every language that can be compiled into the LLVM IR – e.g. C, C++, Java, Ada, Swift or Haskell – by any verifier that is able to work on the BOOGIE IVL – e.g. BOOGIE, DUALITY or CORRAL. It additionally takes advantage of various optimizations applied by compilers when reducing input to the LLVM IR and gives SMACK access to analyses performed by the compiler.

While the LLVM IR has several advanced features – such as dynamic memory, pointer arithmetic, and floating points – BOOGIE IVL is in its very nature restricted to a mathematical program representation [Barnett et al. 2006]. This means that all of those features listed before are not available in BOOGIE and have to be modeled. SMACK does so by means of the unbounded integer data type and arrays thereof that BOOGIE IVL provides. Internally, it applies static analysis to find distinct memory locations such that the dynamic memory does not have to be modeled as a large, single array. This has the advantage of easing the load for the back-end verifiers.

Once the transformation has finished, SMACK passes the BOOGIE IVL program to a verifier of choice. As support for DUALITY is – as of now – experimental, we will focus on the verifiers BOOGIE and CORRAL.

BOOGIE is the default verifier for the same-named verification language [Barnett et al. 2006]. It applies transformations to the input program in a pipe-like style, and finally converts it to a verification condition that is passed to an automatic theorem prover. Among other techniques, the transformation employs abstract interpretation to infer loop invariants and thus program correctness.

CORRAL builds on BOOGIE [Lal et al. 2011]. As noted in 2LS’s drawbacks, invariant generation for arithmetic and arrays can be a hard task. Thus, CORRAL is not based on invariant deduction, but rather bounded model checking. Instead of discovering new states in a breadth-first manner like CBMC does, a goal-directed search algorithm is applied. If procedures can be identified as irrelevant for proving the property, CORRAL does not uncover those program parts. Furthermore, an abstraction over the set of program variables is implemented. Here, a set of tracked variables contains the currently observed part of the program state. This set is refined in a CEGAR approach once a spurious counterexample is identified, i.e. the precision was not sufficient.

In its default configuration, SMACK employs CORRAL in a bounded model checking setting, i.e. it requires a bound being set a-priori, otherwise it will terminate after one loop unrolling.

**Strengths:** Most obviously, SMACK allows for target languages and verifiers to be plugged-in easily, thus leveraging specialized back-end verifiers to more general purposes. SMACK’s infrastructure also reduces the required effort in building a verifier. Rather than re-implementing program semantics every time, one can rely on SMACK’s transformation to the BOOGIE IVL instead. The verifiers in turn can be based on a programming language with simple and clean semantics and few edge-cases. Besides, converging to one framework for program semantics limits the amount of possible errors one can make, compared to any self-implementation thereof. Besides offering a common framework, SMACK also provides performance improvements as compiler optimizations such as constant propagation and control flow graph reduction are readily available.

**Weaknesses:** As only BOOGIE and CORRAL are stably supported, proof generation is very limited in SMACK. Additionally, complex features such as floats and bit-precise operations can not be directly represented in the BOOGIE IVL and have to be modeled

using integers. This leads to a loss of abstraction, which can be fatal for SMT-solvers already supporting such theories. As our case studies rely vastly on both of the before-mentioned features, SMACK’s applicability remains uncertain.

### 3.3.6 CPAChecker

The Configurable Program Analysis Checker (CPACHECKER) provides a framework for implementing various analysis and verification techniques [Beyer et al. 2007]. Besides providing a common framework using the CPA algorithm, it also implements most of the verification approaches presented in section 3.2 [Beyer et al. 2008] [Beyer et al. 2010] [Beyer and Löwe 2013] [Beyer et al. 2015]. It is actively developed by the SoSy-Lab of the LMU Munich and was originally presented in 2009.

In its core lies the CPA algorithm. It is a generalization of a reachability analysis and operates on an abstract representation of a verification procedure, also called *configurable program analysis* (CPA). Any concrete analysis implementing this representation has to provide interfaces that the CPA algorithm can then access during its construction of the reachability graph. Thus, CPACHECKER abstracts away from the actual details of program analysis and model checking and identifies the common ground between them – that is, constructing a reachability graph whose nodes represent abstractions of the actual program semantics. This enables CPACHECKER to cover a wide range of analysis approaches, starting from exact explicit model checking and ending with abstract static analysis.

The workings of CPA are presented in the following. For a more detailed introduction, we refer to [Beyer and Keremoglu 2011]. For a given abstract domain  $D$ , it iteratively uncovers abstract states from the semi-lattice of  $D$ , which is called  $E$ . After its termination, it outputs the set of reachable states from that abstract domain. For this, it keeps a wait list of unexpanded and a set of fully explored states. As long as there is an unexpanded abstract state  $s$  in the wait list, it does the following:

1. Remove  $s$  from the wait list.
2. For each *successor*  $s'$  of  $s$ :
  - a) *Merge*  $s'$  with any already discovered abstract state  $s''$ .
  - b) Check if that leads to a new state, if so replace  $s''$  with the merge result.
  - c) Check if we should *stop* exploring  $s'$ , if not, add it to the state structures.

While this represents a very straight-forward approach to state-space exploration, the actual complexity of the concrete analysis is expressed with the implemented definition of the three operators written in italic above:

- *Successor*: The transition relation from an abstract state to its abstract successor states.
- *Merge*: Given two abstract states, it returns a new abstract state representing the

combined information of both.

- *Stop*: Given an abstract state, checks if it is already covered by the reached states.

To illustrate this approach, a sketch of how those operators would be defined in a static analysis as well as an explicit model checking setting is given in Table 6.

Table 6: An example of the CPA operators.

<i>Operator</i>	<i>Static analysis</i>	<i>Explicit model checking</i>
<i>Successor</i>	All succeeding program locations	All succeeding program states
<i>Merge</i>	States with the same locations	Never
<i>Stop</i>	If the abstract state only introduces already known concrete states	If the concrete states of the given abstract states are already completely represented by another abstract state

This approach has led to a variety of implemented CPAs and combinations thereof, as CPACHECKER supports combining analyses into a composite. This results in an improved trade off between precision and speed, as highly abstracting techniques join more often, but more explicit analyses allow a path-sensitive exploration of the state space. Among others, bounded model checking,  $k$ -induction, explicit CEGAR-analysis, predicate analysis and symbolic execution have been implemented in this framework.

**Strengths:** The greatest advantage of using a framework such as CPACHECKER lies in its versatility. Having implemented basically any promising verification approach of the last decades, it is hardly surprising that a sequential combination of those has placed in the top three overall in the last three SV-COMPs. A sequential application has the advantage of harvesting the complementary strengths of certain approaches. Further, by applying a multitude of configurations from the same framework to a verification task, one eliminates the effect of the environment itself on the observed outcome. Thus, we can directly compare techniques on our case study instead of comparing verifiers.

**Weaknesses:** CPACHECKER contains an incredibly large amount of configuration files. Choosing the a sufficiently good set of verification techniques requires prior knowledge or experimentation, for the case that the default configuration does not deliver the desired results. While theoretically sound, some of the analyses still have certain practical limitations. For example, the predicate analysis does not yet support interpolation over floating point variables.

### 3.3.7 UltimateAutomizer

ULTIMATEAUTOMIZER implements a trace abstraction based on automata in a CEGAR scheme, as presented in section 3.2.9 [Heizmann et al. 2013]. Its development started in 2013 and is based on the ULTIMATE framework which provides unified access to program

representation, automata operations, code transformations, and SMT-solvers.

Similar to SMACK, ULTIMATEAUTOMIZER transforms its input program into the Boogie IVL on which then various optimizations are applied. Finally, the already depicted CEGAR scheme iterates until a sufficient precision is reached. ULTIMATEAUTOMIZER can either be configured with an over-approximation of bit-precise operations by means of mathematical integers, or directly instructed to employ the SMT-solver’s bit vector theory, which is necessary for our case study. Since its original implementation, various improvements – such as a sophisticated interpolation scheme [Heizmann et al. 2018] – have been added.

**Strengths:** Basing the abstraction purely on automata representing error traces can reduce the amount of information that has to be stored by the verifier. This coarse abstraction can lead to an easier proof generation, while still being able to argue about single counterexample paths.

**Weaknesses:** It remains to be shown whether the interpolants derived on our case study are strong enough to actually build “meaningful” error trace automata. If interpolants are too weak, very little information is added in each CEGAR iteration, which in turn can lead to very little spurious counterexamples being excluded. Additionally, deriving interpolants over complex programs that contain arrays, floating point numbers, and pointer arithmetic can be challenging. ULTIMATEAUTOMIZER supports such interpolants since 2015 [Heizmann et al. 2015], meaning that a lot of improvements in this area are still on its way.

### 3.3.8 k-Induction

Alongside the verifiers competing in the SV-COMP, we have implemented a tool that enables plain  $k$ -induction on top of any bounded model checker. Its main goal is harvesting the power of extremely efficient bounded model checkers for proof generation. In this way, verifiers like CBMC or CBMC INCREMENTAL, which natively only support bug hunting but have matured over time, can be elevated. Furthermore, one can take advantage of features such as fine-tuning the bound for each loop in the program.

The implementation of this tool relies on the transformation described in section 3.2.4. As a rough overview, it applies the mentioned code transformations to create a new program representing the induction step. It then runs the back-end verifier on both the base step – i.e. the input file – and the induction step. If the base step returns a counterexample, the tool reports *False*. In case the induction step returns no counterexample for iteration  $k$  and the base case has also reached  $k$ , it reports *True*. Since it is tailored to the specific case study, we do not check the forward case.

The tool is held configurable such that any back-end verifier can be easily plugged in. For this, the user has to define which verifier outputs correspond to which results, and how to call the verifier on the base and induction case. In addition, the tool is able to run on incremental bounded model checkers for further performance improvements. In

a standard bounded model checking setting, the tool creates new verifier instances for every increment of  $k$ .

**Strengths:** The main advantage lies in its configurability, thus enabling proof generation for advanced verifiers. This can be especially useful if we observe that tools supporting  $k$ -induction and bounded model checking do not deliver promising results even on the bounded model checking setting, but plain bounded model checkers do. If such tools are more efficient in the base case, it can be speculated that they also exhibit this efficiency in the induction step. Further, our case study contains loops in its step. Despite being bounded, not every verifier is actually able to infer a bound. When using CBMC with  $k$ -induction, we are able to specifically tell the verifier the bound of the loops contained in the step, meaning that  $k$ -induction is only applied to the main loop.

**Weaknesses:** The tool has currently restrictions on its input code. It is targeted to embedded-style programs containing one main loop with a bounded loop body. The property has to be checked at the very end of every loop iteration. Although there exist transformations from general programs to one-loop programs, we decided to skip this step since our case study do not exhibit nested unbounded loops. Moreover, the  $k$ -induction tool is only as strong as its back-end verifier. This holds especially for its ability to handle large non-deterministic state spaces, as the induction step transformation can havoc a large amount of the program variables. Finally, we do not implement any approaches to strengthen the inductive invariant, as verifiers like CPACHECKER and 2LS do. As seen in section 3.2.4, this can potentially lead to fewer properties being proven.

### 3.4 Leverage points for improvement

Besides employing various verifiers on a variety of approaches, we also identify and analyze possible leverage points to further optimize result coverage and run times. In a typical verification process, one can derive three points on which optimizations can be added:

1. **Properties:** There are various semantically equivalent formal specifications for one requirement. Choosing a form that is easy on the verifier is indispensable, although hard to decide in general. Avoiding expensive look backs and trigger-response patterns are exemplary.
2. **Code:** The input program can be fine-tuned to fit the verification task, for example by ignoring code specifics that are of no interest to the reachability of the safety property. Some code features, such as floating point arithmetic and dynamic memory, affect verifiability to a greater extent. Moreover, the code can be transformed to be more comprehensible for verifiers and thus aiding the verification process.
3. **Verifier:** The efficiency of the verifiers themselves has most likely the greatest total effect. A more efficient verifier can produce more and quicker results. For this, observing the effect of configurations plays a vital role.

It has to be noted that the requirements itself were already prespecified in natural language, and thus not changeable. Property specification was then done manually where the requirements were transformed to formal specifications with efficiency in mind. In the following, we will focus on the second and third leverage point.

### 3.4.1 Code

This work employs *static analysis* for code complexity reduction in order to aid the verification process. Static code analysis is an umbrella term for techniques that only rely on information available at compile time, i.e. on the parse tree [Nielson et al. 1999]. This is in contrast to *dynamic analysis* which is based on applying semantic rules, i.e. interpretation.

One of the most prominent static analysis tools for C is *Frama-C* (Framework for Modular Analysis of C programs) [Cuoq et al. 2012], an open-source framework written in OCaml. It contains a collection of several program analyzers, some of which are employed for the static analysis purposes in this work. It has to be noted that any code produced by Frama-C will be represented by means of an internal abstract syntax tree representation which is a normalization of the input program. Thus, although semantically sound, exported Frama-C code may contain `goto` statements and other constructs not present in the original code. As an industrial-strength alternative, we also considered GrammaTech's *CodeSurfer* [Anderson and Teitelbaum 2001], but found that it was missing an export functionality which is mandatory for feeding the transformed code to the verifiers.

In the following, the three techniques for which effect sizes were measured during benchmarking will be presented.

**Slicing:** Programs may contain pieces of code that are irrelevant to the reachability of a certain program location. Slicing is the process of removing statements that do not influence the reachability of a given program statement [Weiser 1981]. As an example, consider Figure 17.

In our case, the reachability of the `__VERIFIER_error()` call is of special interest. In the example, the calculations over the variable `y` are unnecessary for the value of `x`. The product of a slicing process is called a *slice*, which is presented on the right hand side. Here, any statement containing `y` was removed. This slice is more easily verifiable, as the bug can be found within ten main loop unrollings, in contrast to unrolling the inner loop of the original program to a total of 56 times.

**Value analysis:** The aforementioned slicing process relies on a so-called *value analysis* to determine the interdependency between statements [Canet et al. 2009]. In such a static analysis, value domains of the program variables are over-approximated based on abstract interpretation, with the goal of determining information about their value domain. In most programs, variables seldom exhaust their complete state space, but are rather limited to a certain range.

```

1 extern void __VERIFIER_error();
2 int main() {
3     int x, y = 0;
4     while (1) {
5         x++;
6         while(y < x) y++;
7         y = 0;
8         if (x == 10) {
9             __VERIFIER_error();
10        }
11    }
12 }

```

↔

```

1 extern void __VERIFIER_error();
2 int main() {
3     int x = 0;
4     while (1) {
5         x++;
6
7
8         if (x == 10) {
9             __VERIFIER_error();
10        }
11    }
12 }

```

Figure 17: An Example of slicing, where the left program is unsliced and the right program contains the slice for the reachability of `__VERIFIER_error()`;

```

1 int x, y = 1;
2
3 void step() {
4     x++;
5     if (x == 0) y += 1;
6     if (x == 1) y *= 2;
7     if (x == 2) x = 0;
8
9
10 }
11
12
13 int main() {
14     while (1) {
15         step();
16         if (y * 2 < 0) {
17             return 0;
18         }
19     }
20 }

```

↔

```

1 int x, y = 1;
2
3 void step() {
4     x++;
5     if (x == 0) y += 1;
6     if (x == 1) y *= 2;
7     if (x == 2) x = 0;
8     __VERIFIER_assume(x == 0 || x == 1);
9     __VERIFIER_assume(y >= 1 &&
10                        y <= 2147483647);
11 }
12
13 int main() {
14     while (1) {
15         step();
16         if (y * 2 < 0) {
17             return 0;
18         }
19     }
20 }

```

Figure 18: A program where the state space is restricted to  $x \in 0, 1$  and  $y \geq 0$  at the end of `step()`. The right-hand side shows its transformation after applying information from a value analysis.

The left hand side of Figure 18 shows a program whose variables only use a part of their available domains. It increases `y` depending on the current state `x` until it is susceptible to an overflow in the next iteration. Then, it returns. It follows that `y` can never overflow and thus never be of a negative value. `x` on the other hand is bounded to the finite domain  $\{0, 1, 2\}$ .

Frama-C is able to automatically compute such domains, but only for the last location of each function and not for arbitrary program locations. This is also the reason why the example employs a `step()` function. Frama-C produces the following restrictions at the end of `f()`:

```
[value:final-states]: Values at end of function f:
    x ∈ {0; 1}
    y ∈ [1..2147483647]
```

If we now forward such information to the verifiers, the constructed state space can potentially be reduced. This is of special significance when deriving loop invariants in over-approximating techniques. Some verifiers also internally apply static analysis for such purposes – e.g. 2LS or CPACHECKER – but for those who do not, hints can be of additional value. We do so by fetching the domain over-approximation from Frama-C at the end of each function. Then, we add `__VERIFIER_assume()` statements at those locations containing the calculated domains. For the example program, this transformation is presented in the right hand side on Figure 18, lines eight to ten.

This process was automated by means of a shell script that processes Frama-C’s output and puts the assume statements to the correct locations.

**Variable moving:** We observed a wide range of variables being unnecessarily in a global scope. This situation arises due to the open-loop setting that we observe the case studies in – in the production environment, those variables will be needed in a global scope to be accessed by other system parts. Hence, for verification purposes we apply a relocation of such variables to their function-local scope.

```

1 int x, y = 1;
2
3 void step() {
4
5     x++;
6     if (x == 0) y += 1;
7     if (x == 1) y *= 2;
8     if (x == 2) x = 0;
9 }
10
11 int main() {
12     while (1) {
13         step();
14         if (y * 2 < 0) {
15             return 0;
16         }
17     }
18 }

```

↔

```

1 int y = 1;
2
3 void step() {
4     static int x = 1;
5     x++;
6     if (x == 0) y += 1;
7     if (x == 1) y *= 2;
8     if (x == 2) x = 0;
9 }
10
11 int main() {
12     while (1) {
13         step();
14         if (y * 2 < 0) {
15             return 0;
16         }
17     }
18 }

```

Figure 19: The variable moving process, illustrated on the exemplary program. `y` can be moved to local scope, whereas `x` has to stay global.

Figure 19 present such a process. Here, the variable `y` is only accessed in `step()`. Its declaration can thus be moved to the body of `step()`. Note that it has to be modified to be static, as it is not constant and has to maintain its value over multiple iterations. `x` on the other hand is read both in `step()` and `main()`, hence we can not move `x` due to its presence in the main function. A reduced amount of globals decreases the interconnectivity between functions which can ease the proof finding process.

We developed a tool based on `ctags` and `PyCParser` that does this job in an automated fashion. Global variables are extracted from a C file using `ctags`. In the following, the tool fetches the scopes for each variable, which we then use to change the parse tree such that those variable are moved to their respective local scope, if possible. `PyCParser` allows for an export to C code, which is subsequently used.

### 3.4.2 Verifiers

The configurability of the verifiers depends highly on the tool chain. For example, `CPACHECKER` is heavily configurable, whereas `CBMC` has only little options for possible enhancements. While the effect of configurations was still explored if possible, this section focuses on satisfiability solvers. As every verifier reduces its analysis to either an SAT- or SMT-problem, the end result can be influenced vastly by the solvers. Since they provide a common ground, it is subsequently advisable to analyze this impact.

Table 7: The support of different SAT- and SMT-solvers of the selected verifiers.

Abbreviations: <sup>d.</sup> = default, <sup>exp.</sup> = experimental, <sup>n.a.</sup> = not available, <sup>n.f.</sup> = no floating point theory, <sup>n.w.</sup> = not working.

	CBMC	ESBMC	2LS	SMACK	CPACHECKER	UAUTOMIZER
<i>Boolector</i>	✓ <sup>n.w.</sup>	✓ <sup>d.</sup>				
<i>MathSAT</i>	✓	✓ <sup>n.a.</sup>			✓ <sup>d.</sup>	✓ <sup>d.</sup>
<i>CVC4</i>	✓	✓ <sup>n.a.</sup>		✓ <sup>exp.</sup>		✓ <sup>n.f.</sup>
<i>Yices</i>	✓ <sup>n.w.</sup>	✓ <sup>n.a.</sup>				
<i>Z3</i>	✓	✓		✓ <sup>d.</sup>	✓ <sup>n.w.</sup>	✓
<i>SMT-Interpol</i>					✓ <sup>n.f.</sup>	✓ <sup>n.f.</sup>
<i>Princess</i>					✓ <sup>n.f.</sup>	
<i>Minisat2</i>	✓ <sup>d.</sup>					
<i>Glucose</i>			✓ <sup>d.</sup>			

Since 2003, *SMT-Lib* aims to provide a common interface to SMT-solvers, and is currently available in its second iteration [Barrett et al. 2010]. Most verifiers employ such an interface to allow for an exchange of back-end solvers. We identified a total of nine distinct solvers used by the selected verifiers, two of which are SAT-solvers.

Table 7 presents the support of available solvers for the selected verifiers. Here, the first six rows correspond to SMT-solvers, where the last two represent the SAT-solvers used

by CBMC and 2LS. The default configuration of each verifier is marked with <sup>d.</sup>. Some combinations of verifiers and solvers could not be applied to our specific case study: Few did just not work correctly – i.e. produced crashes or incorrect behavior. This is indicated by <sup>n.w.</sup> (not working). In other cases, the floating point theory was not (yet) supported (<sup>n.f.</sup>, no floating point theory). ULTIMATEAUTOMIZER especially relies on interpolation over floats, which also falls into this category. In the case of ESBMC, solvers had to be compiled into the project, which we tried with no success (<sup>n.a.</sup>, not available). SMACK only supports Z3 as its back-end solver, where CVC4 support is still experimental and thus excluded from this study.

Hence, we identified the following additional non-default solvers as potential leverage points for increasing result coverage during benchmarking:

- CBMC: MathSAT, CVC4, Z3
- ESBMC: Z3
- ULTIMATEAUTOMIZER: Z3, MathSAT

## 4 Experimental setup

This section presents the experimental setup of the conducted study with an emphasis on the ability of reproducing the verification runs.

### 4.1 Experimental environment

All experiments were performed on a machine with four AMD Opteron 6172 processors, each containing twelve cores at 2.1 GHz. A total memory of 189 GB was available. We executed ten benchmarks in parallel, giving each execution four CPU cores with a memory limit of 18 GB. The time limit was set to two hours of CPU-time for each verification task.

For every verifier configuration running on a given specification, we collected the following data points:

- The exact command that was run.
- The specification task of the run.
- The result; either *True*, *False* or *Unknown*.
- The reason if no definite answer could be given, such as timeout, memory overflow or the presence of a bug.
- The used CPU-Time, in seconds.
- The peak memory usage, in MB.
- If measurable, the time spent in a satisfiability solver, in seconds.
- If measurable, the reached depth in a BMC or  $k$ -induction setting.

Recall that some requirements were split into multiple formal specifications. Thus, the collected specification data was reflected back on the actual requirements. The verifier specific data points – such as CPU-time and memory – can not be fairly transferred to the requirements since their actual performance was not measured on that specific task. We therefore only record the final results on the requirements.

### 4.2 Code setup

In this part, we describe the technical process of producing a verifiable C code file for each requirement. Originally, the code that resulted from the MatLab export was given by Ford. It contained the following parts:

- A `main.c` file containing the code, as well as three header files: one for compile time consistency checks, one for private, and one for extern declarations and definitions.
- An `Include` folder containing the library headers and code.

We created these additional files on our end:

- A `boiler-plate-header.c` file containing defines and functions used by the specifications, e.g. logical operators, truth values, or floating point comparison. Further, it includes the external declarations for the SV-COMP functions, such as `extern void __VERIFIER_error();`
- A `boiler-plate-footer.c` file containing the aforementioned `updateVariables()` function and a stub of the main entry function.
- An `exported_specifications` folder containing the exported specification files from the Pattern Based Specification Tool.

To prepare those files, for every verification task `exported_specifications/spec.c`, the following steps were conducted:

1. Prepend `boiler-plate-header.c` to `main.c`.
2. Append `boiler-plate-footer.c` to `main.c`.
3. In `main.c`, replace the main function stub with `exported_specifications/spec.c`.
4. In `main.c`, replace `step()` with the correct identifier of the case study's actual step function.
5. Apply the compiler's preprocessor to the resulting code, including the `Include` folder, e.g. via `gcc -Wall -E -IInclude main.c`.
6. Remove all comments, as they are not necessary for the verification process.
7. Lastly, remove all `extern` keywords from global variable declarations, as we do not link against the corresponding external declarations.

The preprocessing step is necessary as some verifiers require preprocessed code and are not able to directly run on arbitrary C projects. The process was automated in a batch assembly mode, creating one verification task for each exported specification. At this point, the basic verification tasks are assembled, and a verifier can be run on the output code.

### 4.3 Verifier setup

After assembling the verification tasks, the verifiers were executed. Some were benchmarked with multiple configurations, e.g. due to various supported solvers or verification techniques. In total, we tested 22 configurations of eight verifiers.

CBMC version 5.8: In an early phase, we tested how deep CBMC could unroll the main loop in the given time limit, which yielded a bound of around 20. As CBMC allows fine-tuning the loop bounds, we applied the limits only to the main loop, whereas the remaining loops were set to a bound of 256, which was manually determined to be the limit. Additionally, the small script `cbmc.sh` determines the main loop identifier, as possible `gotos` introduced by Frama-C can lead to multiple loops in the main function. The script takes parameters that should be passed to CBMC, and replaces the `X` in

`main.X`: with the correct main loop identifier number.

Since CBMC does not support an incremental approach, we implemented a small wrapper script `cbmc-incremental.sh` to call CBMC with increasing bounds. As early results have shown that calling CBMC incrementally leads to great performance improvements, SMT-configurations were only tested with the incremental mode.

- `cbmc.sh -32 -unwindset main.X:20 -unwind 256 input.c`
- `cbmc-incremental.sh input.c`
- `cbmc-incremental.sh -z3 input.c`
- `cbmc-incremental.sh -cvc4 input.c`
- `cbmc-incremental.sh -mathsat input.c`

CBMC INCREMENTAL version 5.4: As the objective of including CBMC INCREMENTAL is comparing a wrapped and a direct incremental implementation, only one configuration was benchmarked. Again, `cbmc-incremental.sh` is a wrapper script to determine the correct main loop identifier.

- `cbmc-incremental.sh -32 -incremental-check main.X -unwind 256 input.c`

ESBMC version 5.1.0: ESBMC was benchmarked with two different verification techniques – incremental bounded model checking and  $k$ -induction – and one additional SMT-solver – Z3. Although ESBMC claims that its native SMT floating point mode is a default, we observed different results with and without setting the corresponding option, `-floatbv`. We also set `-unlimited-k-steps` to define the maximum loop bound to its largest value.

- `esbmc -32 -floatbv -incremental-bmc -unlimited-k-steps input.c`
- `esbmc -z3 -32 -floatbv -incremental-bmc -unlimited-k-steps input.c`
- `esbmc -32 -floatbv -k-induction -unlimited-k-steps input.c`
- `esbmc -z3 -32 -floatbv -k-induction -unlimited-k-steps input.c`

2LS version 0.6.0: As early tests revealed that significant results were only achieved with incremental bounded model checking,  $k$ -induction and  $k$ -induction  $k$ -invariants, we restricted 2LS to those configurations.

- `2ls -32 -incremental-bmc input.c`
- `2ls -32 -havoc -k-induction input.c`
- `2ls -32 -k-induction input.c`

SMACK version 1.9.0: SMACK is not able to execute in an incremental mode on our case studies, i.e. terminates after one unrolling if no loop bound is given. After initial tests to determine which loop bounds the time limit allows for, we found that SMACK could only unroll to depth one at most. To make sure this was no sampling error, we additionally included depth three. All tests were run with the CORRAL verifier, as it builds upon

BOOGIE and is the default anyways. A time limit had to be passed to SMACK, as it otherwise exits after a default of 1200 seconds. We additionally need to enable SMACK's floating point and bit precise modes to ensure a correct handling of the verification tasks.

- `smack -float -bit-precise -time-limit 7200 -unroll 1 input.c`
- `smack -float -bit-precise -time-limit 7200 -unroll 3 input.c`

CPACHECKER version 1.7: As quite a lot of configurations are supported – by default, CPACHECKER ships with over 260 configuration files – we decided to run the ones that were chosen for the SV-COMP. This included incremental bounded model checking with  $k$ -induction, a value analysis with a CEGAR approach, and a predicate analysis. To ensure that we did not accidentally leave out a promising configuration, we tested CPACHECKER on several other analyses, including plain bounded model checking, value analysis with a weighted traversal, predicate analysis with adjustable block encoding, and symbolic execution with and without CEGAR and slicing. We found that none of those configurations delivered results that were not already covered by the three analyses selected for the SV-COMP. Due to this redundancy, we do not cover those results in the upcoming sections. Note that any analysis over linear arithmetic only can not be applied to the case studies, as they contain a fair amount of nonlinear computations.

Similar to SMACK, CPACHECKER also needs a time limit, otherwise it falls back to its default. Furthermore, the memory has to be set explicitly, as the Java Virtual Machine runs with a default heap size of 1200 MB. To extract the time spent with SMT-solving, we add `-stats`. Finally, we set the property specification to `CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )`, contained in the file `scripts/reach.prp`, instructing CPACHECKER to check against reachability of the error location.

- `cpa.sh -stats -32 -timelimit 7200 -heap 18G -spec scripts/reach.prp -bmc-induction input.c`
- `cpa.sh -stats -32 -timelimit 7200 -heap 18G -spec scripts/reach.prp -predicateAnalysis input.c`
- `cpa.sh -stats -32 -timelimit 7200 -heap 18G -spec scripts/reach.prp -valueAnalysis input.c`

ULTIMATEAUTOMIZER version 0.1.23: By default, we run ULTIMATEAUTOMIZER using the reachability tool chain `AutomizerReach.xml`, and the setting for 32-bit reachability with precise bit vector arithmetic, `svcomp-Reach-32bit-Automizer_Bitvector.epf`. Setting the memory model to `HoenickeLindenmann_Original` ensures a precise representation of the memory in the Boogie IVL program.

In its default setting, ULTIMATEAUTOMIZER uses the *WOLF* refinement strategy, choosing interpolants from different solvers in a dynamic scheme. We tested this dynamic selection against using fixed solvers, one configuration with MathSAT and one with Z3. For those, we created two new setting files `svcomp-Reach-32bit-Automizer_Bitvector_z3.epf` and `svcomp-Reach-32bit-`

Automizer\_Bitvector\_mathsat.epf, where the command for the external solver is adjusted to either `mathsat -unsat_core_generation=3` or `z3 SMTLIB2_COMPLIANT=true -smt2 -in`, and the refinement strategy is removed.

- `java -Xmx18G`  
`-jar plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar`  
`-data data -tc config/AutomizerReach.xml`  
`-s config/svcomp-Reach-32bit-Automizer_Bitvector.epf`  
`-cacsl2boogietranslator.entry.function main`  
`-cacsl2boogietranslator.memory.model HoenickeLindenmann_Original`  
`-i input.c`
- `java -Xmx18G`  
`-jar plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar`  
`-data data -tc config/AutomizerReach.xml`  
`-s config/svcomp-Reach-32bit-Automizer_Bitvector_z3.epf`  
`-cacsl2boogietranslator.entry.function main`  
`-cacsl2boogietranslator.memory.model HoenickeLindenmann_Original`  
`-i input.c`
- `java -Xmx18G`  
`-jar plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar`  
`-data data -tc config/AutomizerReach.xml`  
`-s config/svcomp-Reach-32bit-Automizer_Bitvector_mathsat.epf`  
`-cacsl2boogietranslator.entry.function main`  
`-cacsl2boogietranslator.memory.model HoenickeLindenmann_Original`  
`-i input.c`

**K-INDUCTION:** The **K-INDUCTION** tool was developed to be a basic feasibility study, with the main goal to equip **CBMC** with *k*-induction. Besides, any other verifier supporting bounded model checking already contains a *k*-induction mode. Thus, we only plugged in **CBMC** as its back-end, which the tool already provides configuration files for. Additionally, the time spend in SMT-solving was measured via `-smt-time`.

- `python3 src/kinduction.py -config config/cbmc.yaml -smt-time`

#### 4.4 Examined leverage points

For each examined leverage point, we created a new set of verification tasks. Those were assembled in the following manner, starting with the basic files of Section 4.2 and executed in the stated order, building on one another.

1. **Variable moving:** We move the variables to their most local scope by calling the variable scope analysis tool on `main.c` via `python3 analysis.py main.c global_vars.txt main_moved.c`, where `global_vars.txt` contains a list of all global variables that was extracted with `ctags -x -c-kinds=v -file-scope=no`

```
main.c | grep -v volatile | awk 'print $1;' > global_vars.txt.
```

- 2. Slicing:** `frama-c -slice-calls "__VERIFIER_error" main_moved.c -then-on 'Slicing export' -print -ocode main_moved_sliced.c` calls Frama-C with its slicing plug-in. Since Frama-C removes statements with presumably no effects, it also deletes any conditions inside of `__VERIFIER_assume()` calls, as they are present in the `updateVariables()` function. Thus, we implemented a small script adding those statements back from the original files after the slicing procedure.
- 3. Value analysis:** The shell script that extracts the results from the Frama-C value analysis and inserts them into the corresponding functions is called via `./valueanalysis.sh main_moved_sliced.c`. Internally, it calls Frama-C with `frama-c -val main_moved_sliced.c`.

To avoid an extensive blow up of benchmark times – each leverage point has to be tested against each configuration of each verifier – we used *random sampling* of the verification tasks. As early test runs have shown that slicing is in fact effective in reducing the program size for some requirements, and variable-moving is applied to the unsliced code, we decided to run the full evaluation on the variable-moved and sliced verification tasks.

The remaining three verification task configurations – plain; variable-moved but not sliced; variable-moved, sliced and value analyzed – then needed to be sampled. This sampling was done after the verification results for the main task set was gathered. To ensure that the sample remains expressive for each verifier and does not lead to only *Unknown* results, the sampling was done on a per-verifier basis, depending on the main results.

The samples reflect the relation of definite true and false results of each verifier – if the main benchmark contains  $t$  true,  $f$  false, and  $u$  unknown results, then the samples of size  $s$  contain  $t_s = \frac{t}{t+f} \cdot s$  true results,  $f_s = \frac{f}{t+f} \cdot s$  false results and  $u_s = \min\{0, s - f_s - t_s\}$  unknown results. Intuitively, the proportionality between true and false results is reserved, and if not enough definite answers were given, the remainder is filled up with unknown results. Apart from fixing the amounts a-priori, the actual verification tasks were then selected randomly. We set  $s$  to 50, thus selecting around a quarter of the original 208 tasks.

Note that this process restricts the comparability of the sample sets: One can not directly compare samples of different verifiers against each other as the sample selection differs between verifiers. The sampling process was only used to infer the effect sizes of the leverage points *for each verifier*.

## 5 Results

The following section presents the results of the verifier benchmarks. If not stated otherwise, data is displayed in a merged fashion – meaning that both ECC and DSR are joined together, and for each verifier and verification task, the best run over all configurations is chosen.

For some measures, we present statistical significance values and effect sizes. A significance value, denoted as  $p$ , indicates how likely the measured results are transferable to a generalized population, of whom the collected samples are representative. When calculated on measures that were taken from all tasks, this allows to estimate the likelihood of the results also holding for a similar, bigger set of tasks. In other cases, when results are presented on sampled tasks, one can use the significance to deduce whether the sampled measurements are also valid for the complete set of tasks. If one interprets a significance value  $p$  as a probability, one can infer the chance of the examined hypothesis to be invalid. If such a probability becomes low enough, one can reasonably assume the validity of the measurement for the generalized population. Typically, this probability is set to 0.05. Furthermore, the effect size is given as Cohen’s  $d$  (for continuous metrics) and Cohen’s  $h$  (for categorical metrics). Both state the magnitude of the effect on an interval scale, starting with 0 representing no measurable effect. One can in general say that a value of 0.2 corresponds to a small, 0.5 to a medium, and 0.8 to a large effect [Wassertheil and Cohen 1970].

### 5.1 Overall results

For each verifier, we calculated a score depending on their results, as presented in Table 8.

Table 8: The final scores of the selected verifiers.

Verifier	Normalized Overall	ECC	DSR
K-INDUCTION	<b>199</b>	<b>121</b>	<b>74</b>
2LS	<b>93</b>	<b>107</b>	8
CPACHECKER	<b>65</b>	38	<b>25</b>
ESBMC	62	<b>58</b>	12
CBMC	62	32	<b>26</b>
ULTIMATEAUTOMIZER	38	16	17
SMACK	1	1	0

The score is gathered comparable to the SV-COMP, although we do not incorporate witness verification, thus reduce our scoring system to the cases of either giving a counterexample – +1 point – or providing a proof – +2 points. We present the raw scores for both case studies, and a normalized overall score where the result is averaged over the

number of verification tasks in both sub-categories, again similar to the SV-COMP<sup>3</sup>.

K-INDUCTION places first in both ECC and DSR, with its greatest margin coming from the DSR case study. 2LS takes the second place in both overall and ECC, but falls behind on the DSR case study, where CBMC is able to achieve a second place. Due to its results on DSR, CPACHECKER is able to place third overall.

Figures 20 and 21 present quantile plots of the overall and single case study scores, similar to the plots from the SV-COMP [Beyer 2017]. For this plot, the results are sorted by CPU-time, from fastest to slowest. Then, for each result at position  $n$ , the accumulated score up to this verification task is plotted against its CPU-time.

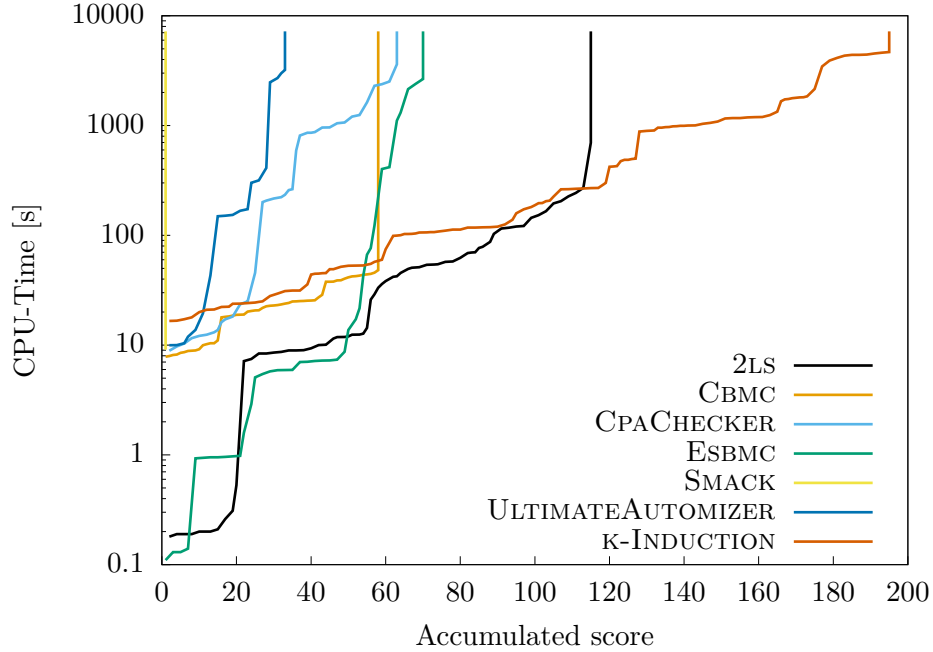


Figure 20: The quantile plots for the overall results.

We observe that all CBMC-based  $k$ -induction and bounded model checking tools, i.e. CBMC, K-INDUCTION, 2LS, and ESBMC, perform comparable up to a score of around 60 points, where CBMC and ESBMC start to diverge. 2LS and K-INDUCTION continue on its exponential path until 2LS stops delivering results at around 120 points. CPACHECKER and ULTIMATEAUTOMIZER exhibit increased CPU-times for low scores already, although CPACHECKER is able to catch up with CBMC and ESBMC for higher scores. The described pattern repeats for the ECC case study, but changes for DSR. Here, ULTIMATEAUTOMIZER and CPACHECKER show improved performance, while 2LS and ESBMC give up early.

In total, the verifiers were able to find a counterexample on 56 and provide a proof on

<sup>3</sup><https://sv-comp.sosy-lab.org/2019/rules.php>

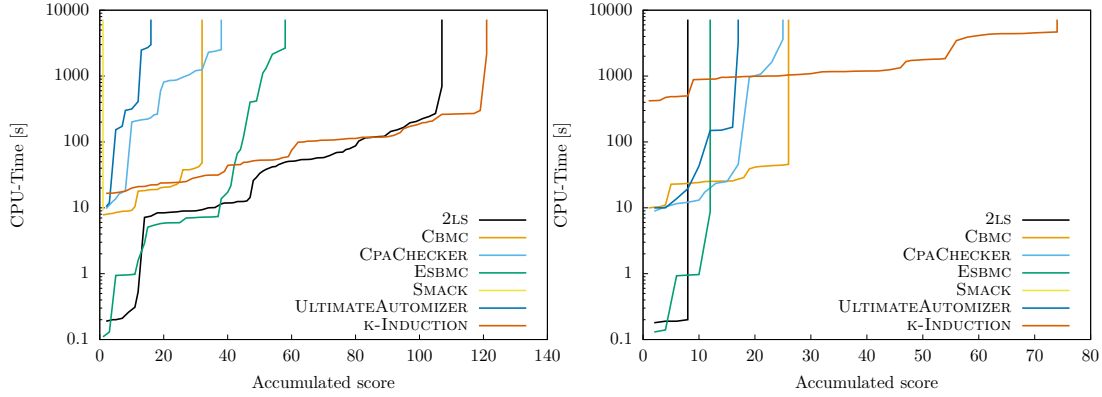


Figure 21: The quantile plots for the ECC and DSR case studies.

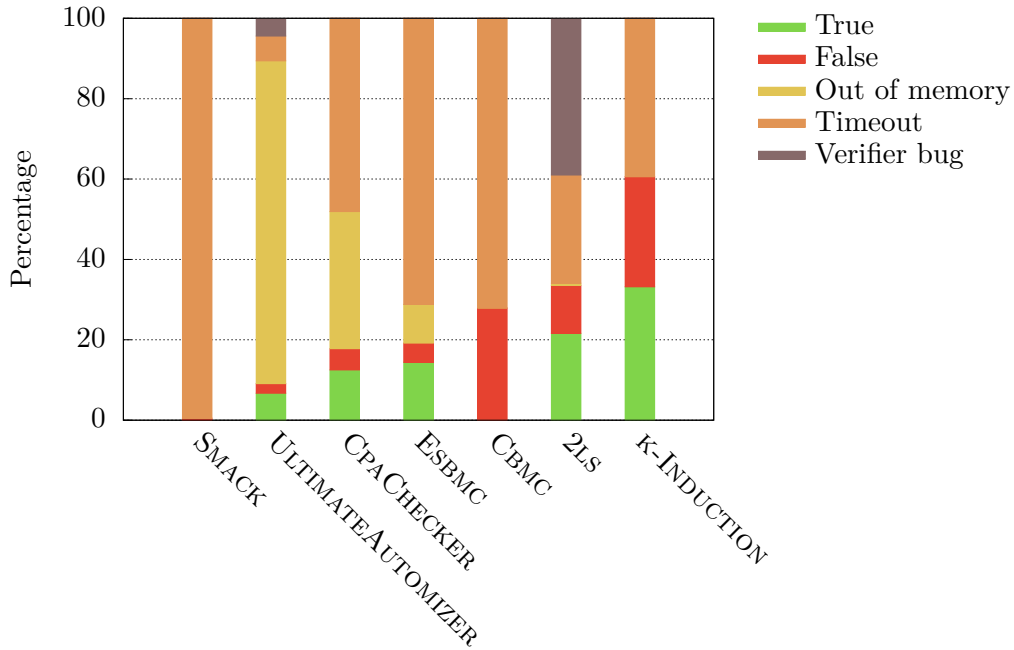


Figure 22: The overall results for each verifier with details for the reasons of indefinite answers.

74 verification tasks. On the remaining 78 files, no answer was given by any verifier. Figures 22 and 23 shows the distribution of answers among the verifiers. It also displays reasons for indefinite results, where the time out of two hours, the memory limit of 18 GB was reached, or the presence of a bug prevented normal execution. We present detailed results of each verifier configuration in the upcoming subsections.

Lastly, the overall location of the verifiers' results in the set of all verification tasks is

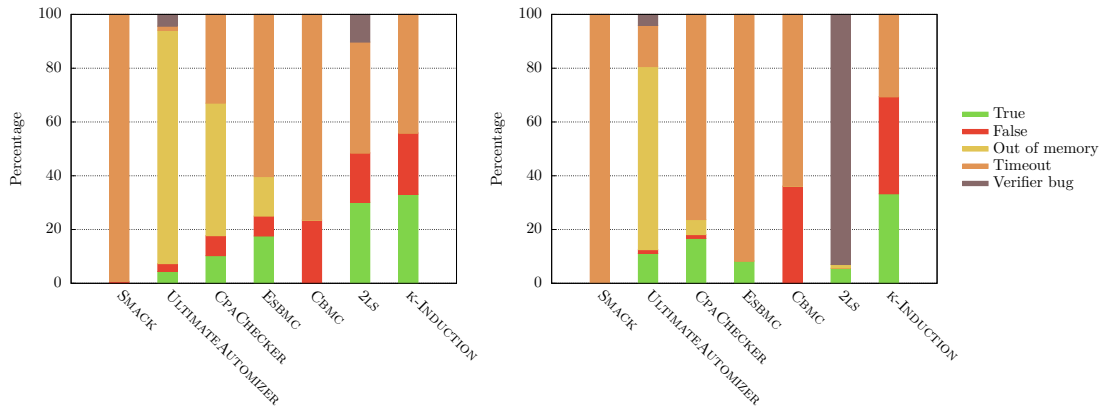


Figure 23: The overall results for each verifier with details for the reasons of non-definite answers for the ECC and DSR case studies.

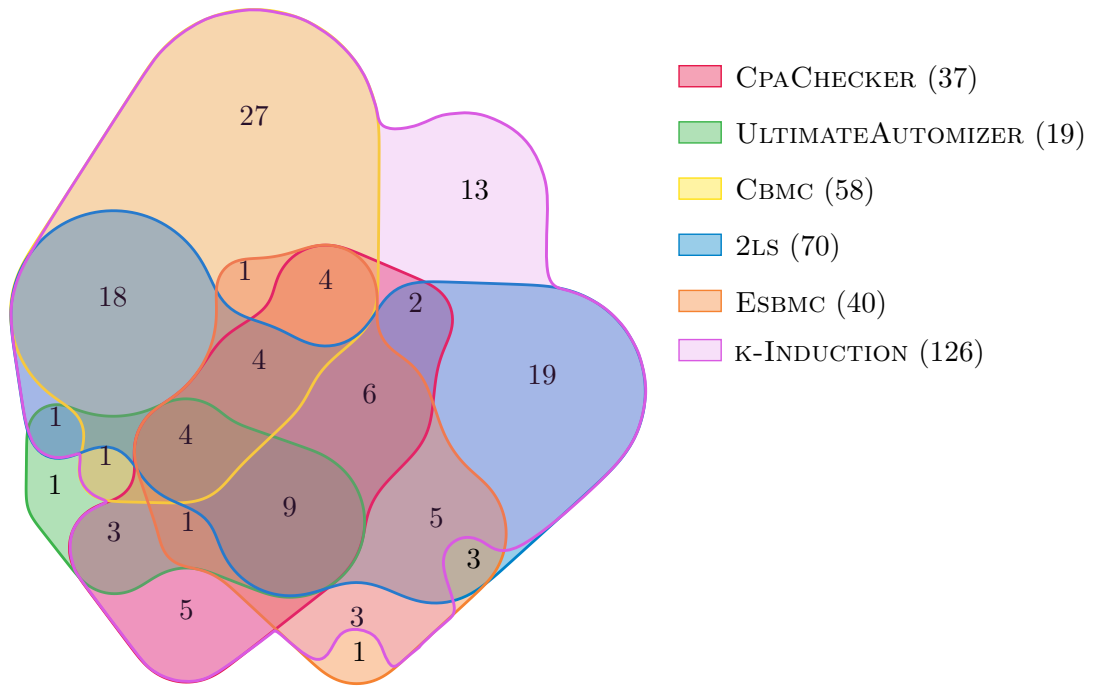


Figure 24: The distribution of definite results for the verification tasks onto the verifiers. The numbers beside each verifier name state its total number of definite results.

presented in form of an Euler's diagram in Figure 24. It was created using the implementation of [Pérez-Silva et al. 2018]. Here, each area that is enclosed by a verifier represents exactly the verification tasks on which the verifier delivered a definite answer.

Those verifier areas are again divided into multiple parts that are marked with numbers. Such a part is covered by one or more verifier areas, and indicates that results on the represented verification tasks were identified exactly by those covering verifiers. The number inside the part states the count of its results.

All but one counterexamples were identified by CBMC, where the remaining violation was found only by ULTIMATEAUTOMIZER. When not considering K-INDUCTION, CPACHECKER, 2LS, and CBMC are able to cover over 90% of all results. K-INDUCTION finds an additional 6% that no other verifier was able to, and comprises 99% of all answers. We find that 85% of ULTIMATEAUTOMIZER’s results were covered by CPACHECKER. 2LS covers 75% of ESBMC’s answers, which in turn finds only 46% of 2LS’s results.

## 5.2 Verifier results

In the following, we present the fine-grained results of each verifier and their configurations. Besides, issues and bugs that occurred during the benchmark process are listed. To aid readability, some data is collectively presented at this point and not in the subsection of each verifier.

Table 9: The absolute memory usage of the verifiers, split into definite (“True”, “False”) and indefinite results (“Unknown”), and relative memory usage over all runs.

Verifier	CBMC	ESBMC	2LS	SMACK	CPA CHECKER	ULTIMATE AUTOMIZER	K- INDUCTION
<i>Memory on def. results [GB]</i>	0.69 ± 0.39	0.55 ± 0.92	0.45 ± 0.27	0.11 ± 0.00	3.00 ± 4.05	2.67 ± 5.01	0.69 ± 0.39
<i>Memory on indef. results [GB]</i>	2.77 ± 0.76	4.69 ± 4.80	0.86 ± 1.63	3.48 ± 1.15	7.60 ± 5.94	14.51 ± 5.13	2.78 ± 0.76
<i>Overall relative memory [MBs<sup>-1</sup>]</i>	2.91 ± 3.76	6.58 ± 32.73	48.27 ± 46.83	1.73 ± 10.37	3.57 ± 6.54	5.32 ± 5.72	2.91 ± 3.76

Table 9 shows the collected statistics for the memory consumption of the verifiers. Here, the measurements are divided on definite (“True” or “False”) and indefinite (“Unknown”) results. Furthermore, the relative memory for all runs is stated. For each metric, the mean and standard deviation is given.

### 5.2.1 CBMC

For CBMC, we report the depths at which the bugs were found. Those are stated in Table 10 for each case study, and as a joined result. We observed most counterexamples occurring at depth two, and half of the remaining violations at both depth one and three. There was no counterexample that could be found at depth four or greater.

In Figure 25, we compare the CPU-times of CBMC to the two other bounded model checking configurations – 2LS and ESBMC incremental. Those scatter plots indicate at

Table 10: The depths at which the bugs were found by CBMC.

Depth	ECC		DSR		Total	
1	3	12%	10	32%	13	23%
2	14	56%	14	45%	28	50%
3	8	32%	7	23%	15	27%

which time a result was found for both verifiers, CBMC’s times being on the abscissa and its comparison on the ordinate. Tasks on which one verifier did not return a result are put onto the *TO* (timeout) indicator on the outer part of the corresponding axis. Cases where both verifiers could not reach an answer are excluded from the plot. Note that for 2LS, the incremental setting actually performs  $k$ -induction  $k$ -invariant, and thus leads to the presence of “timeouts” on CBMC due to its inability to generate proofs.

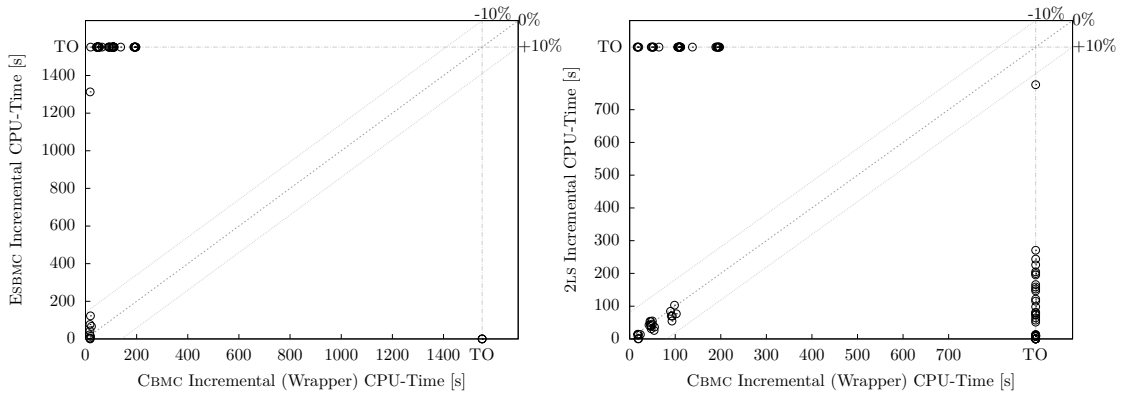


Figure 25: Comparison of CPU-Times of CBMC against ESBMC and 2LS, both in an incremental setting.

In comparison to ESBMC and 2LS incremental, CBMC found around 5.8 respectively 2.3 times more counterexamples. As stated previously, CBMC was able to find all but one counterexamples that were identified by all verifiers combined.

Figure 26 presents a histogram of the CPU-times for CBMC’s definite answers. Most results were found at minutes one and two, with the longest run time being three and a half minutes.

The previously presented data was collected using the incremental wrapper script. When comparing this approach to using a fixed loop bound of 20 iterations, we find that CPU-times decreases from an average of  $4955s \pm 2540s$  to  $99s \pm 61s$  and result coverage increased by 322% – 18 versus 58 definite answers.

**Bugs and issues:** We encountered a bug that presented itself on the code outputted by Frama-C. Such code lead CBMC to report spurious counterexamples. We were able to reduce the issue to a minimal example by employing C-Reduce [Regehr et al. 2012]. In

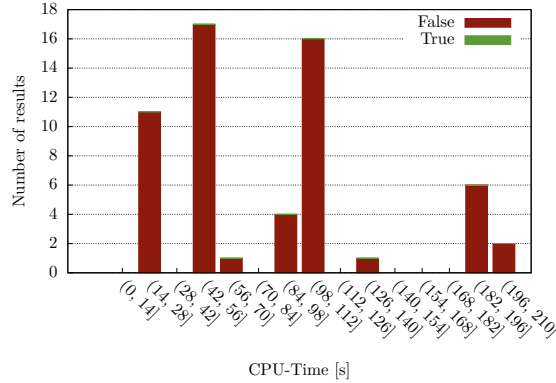


Figure 26: The CPU-times of CBMC at which results were found.

its essence, CBMC does not handle variables that are local to a switch block correctly and assumes a non-deterministic value for them, even if such a variable is set deterministically in one of the `case` blocks. For further details, we refer to the corresponding bug report<sup>4</sup>. No fix was implemented to the point of writing, leading to an inability to test the effect of static analysis for CBMC.

### 5.2.2 CBMC Incremental

Figure 27 compares CBMC INCREMENTAL to plain CBMC with an incremental wrapper script. Where the original has a mean CPU-time of  $84.0 \text{ s} \pm 55.8 \text{ s}$ , the incremental branch reduces it down to  $24.0 \text{ s} \pm 12.2 \text{ s}$ . This effect is significant ( $p = 0.000 < 0.05$ ), as yielded by a paired sample  $t$ -test [Student 1908]. Result coverage was observed to be exactly the same.

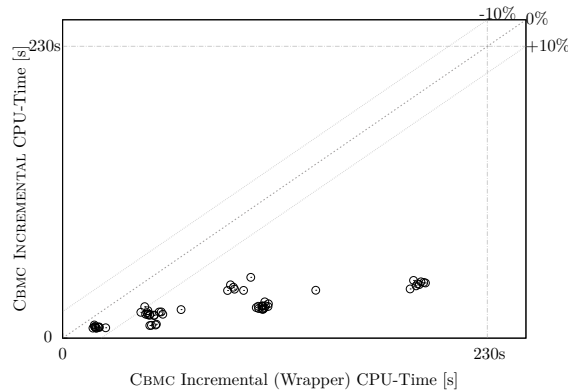


Figure 27: Comparison of CPU-times of CBMC against CBMC INCREMENTAL.

<sup>4</sup><https://github.com/diffblue/cbmc/issues/3283>

CBMC INCREMENTAL was able to find its results in no more than a minute and as early as seven seconds, as presented in the time histogram of Figure 28.

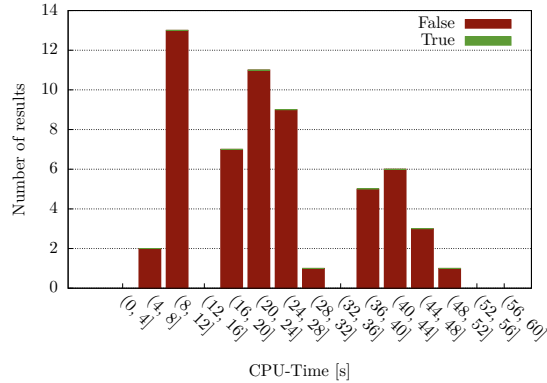


Figure 28: The CPU-times of CBMC INCREMENTAL at which results were found.

### 5.2.3 ESBMC

Figure 29 gives a comparison of ESBMC to all other tools using  $k$ -induction – 2LS, CPACHECKER in a  $k$ -induction configuration and the K-INDUCTION tool. When not considering the time outs, one can observe that 2LS was able to find proofs in a matter of minutes, whereas ESBMC’s  $k$ -induction could take up to 40 minutes. On the other hand, ESBMC shows faster results in comparison to CPACHECKER. For K-INDUCTION, both tools either deliver a result in around the same time, or one tool takes significantly longer whereas the other finds an answers quickly.

The result coverage shows that 33 results were found by 2LS that were not found by ESBMC, but there are also 44 tasks on which ESBMC but not 2LS returned successfully. The result coverage of K-INDUCTION encompasses all but three proofs found by ESBMC.

**Bugs and issues:** For ESBMC, we identified at least four instances in which the default configuration using Boolector delivered conflicting results with comparison to the other verifiers. Specifically, ESBMC gave  $k$ -induction proofs where the majority of verifiers agreed upon a counterexample. Moreover, even ESBMC’s base step identifies the bug, but is stopped due to the induction step providing a proof fast than the base step finds the bug. This internal discrepancy hints to a bug in the application of the  $k$ -induction technique. Thus, we excluded the Boolector configuration from further analysis, as those problems could not be observed with Z3. All of the presented analyses represent the Z3 configuration.

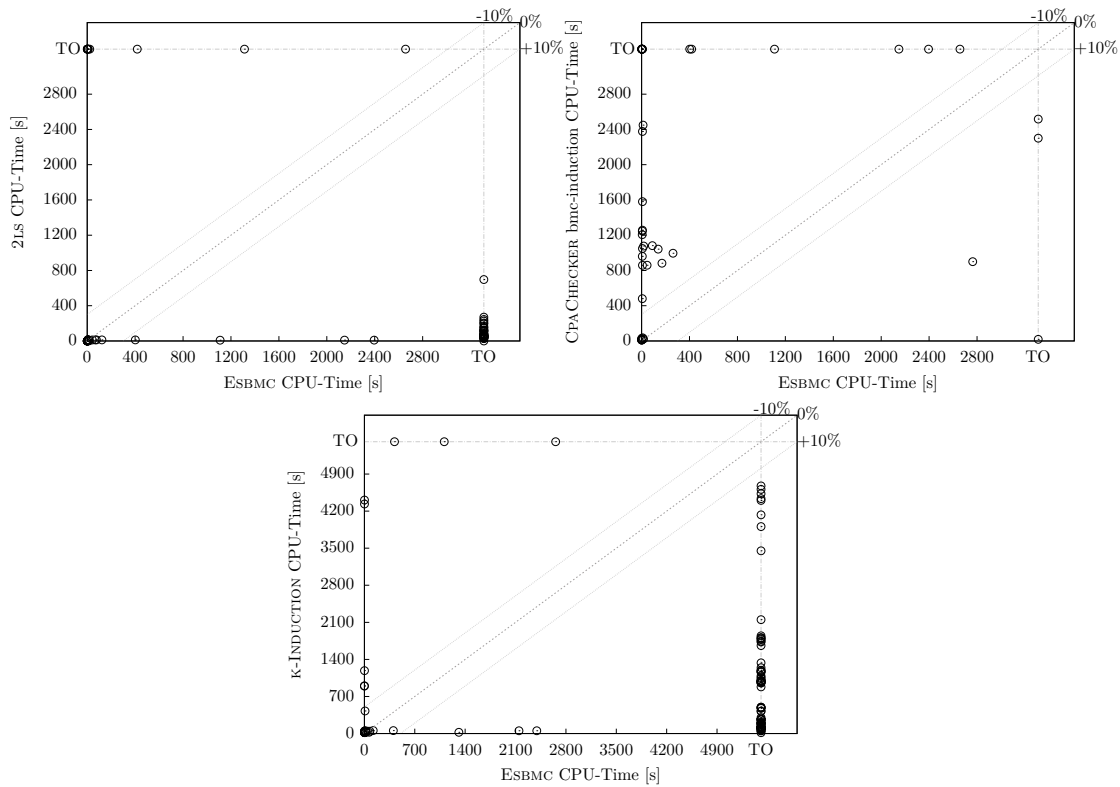


Figure 29: Comparison of CPU-times of ESBMC against 2LS, CPACHECKER, and K-INDUCTION, all in their  $k$ -induction setting.

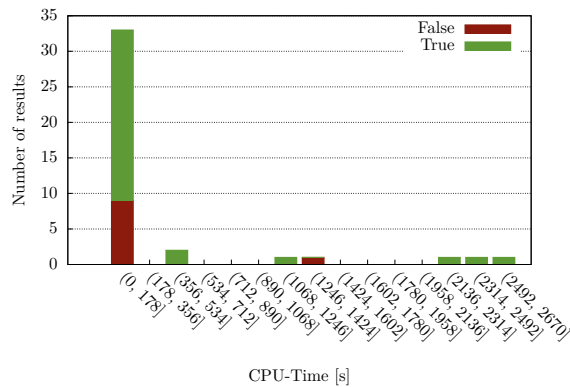


Figure 30: The CPU-times of ESBMC at which results were found.

### 5.2.4 2LS

For 2LS, we examined three configuration:  $k$ -induction,  $k$ -induction  $k$ -invariant and incremental bounded model checking. We observed that standard  $k$ -induction finds

79.5% more results than the  $k$ -invariant version, which in turn was not able to find any result that was not covered by  $k$ -induction already. We found the incremental bounded model checking covering exactly the same results as  $k$ -induction, indicating that this setting was internally enhanced with a  $k$ -induction proof generation.

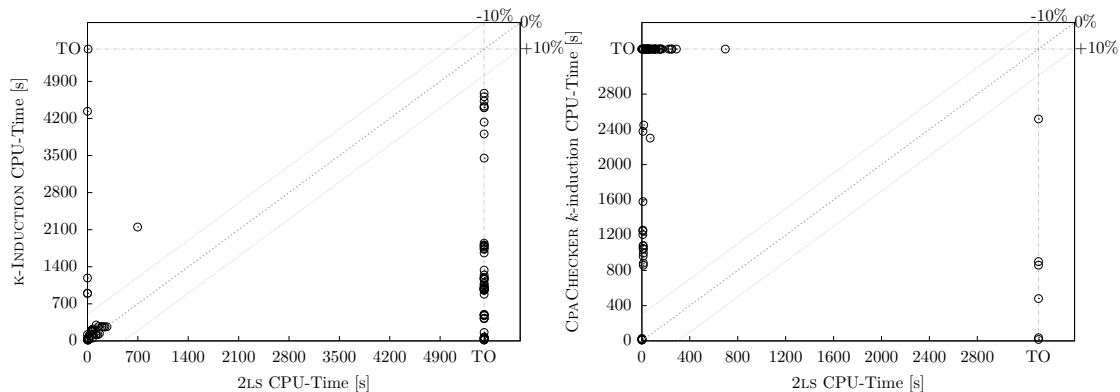


Figure 31: Comparison of CPU-times of 2LS against K-INDUCTION and CPACHECKER with a  $k$ -induction setting.

We compare 2LS against the K-INDUCTION tool and CPACHECKER with a  $k$ -induction configuration in Figure 31. 2LS shows similar performance to K-INDUCTION when disregarding its time outs, and performs considerably faster than CPACHECKER.

Figure 32 presents the run times of 2LS. In over half of the cases, 2LS was able to find a result in less than a minute. For the remainder, an answer was given in less than six minutes, except for one task that was solved after around twelve minutes. If a property violation was present, a counterexample was identified in fewer than two minutes.

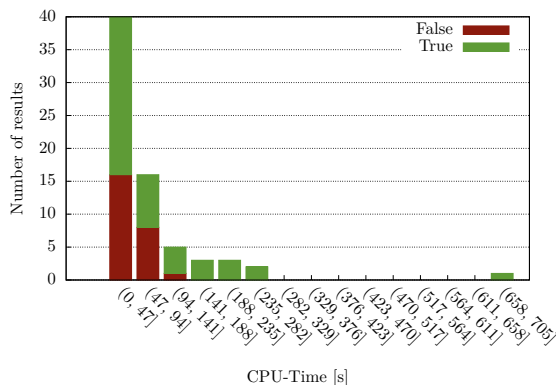


Figure 32: The CPU-times of 2LS at which results were found.

**Bugs and issues:** We identified a simple program on which 2LS delivers false negatives, consisting of two nested loops and a `__VERIFIER_error()` statement after the inner loop. 2LS reports such a program as safe with its  $k$ -induction setting. We refer to

the corresponding bug report for more details<sup>5</sup>, although up to the point of writing, no response was given. Additionally, due to a failed type check in the internal float bit-vector expression simplification, 2LS was not able to run on the DSR case study at all. The same bug prevented 2LS from running on the unsliced verification tasks of the ECC case study, although this bug was not present on the sliced ECC tasks. Finally, 2LS refused to operate on 14 Frama-C outputs of the ECC case study on which it states that an irreducible control flow is present.

### 5.2.5 SMACK

SMACK was only able to deliver a definite answer on one of the 208 verification tasks – a counterexample that was identified after ten seconds. On the remainder, time outs were observed after two hours CPU-Time. Subsequently, we exclude SMACK from our analyses. It has to be noted that SMACK was in fact able to deliver more results when increasing the time limit to around six to ten hours, but this was out of the scope of our experiments.

### 5.2.6 CPAChecker

Figure 33 compares the three configurations of CPACHECKER against each other, as well as their combined results against ULTIMATEAUTOMIZER. On the tasks that both were able to solve, ULTIMATEAUTOMIZER and CPACHECKER exhibit comparable performances, with a slight advantage for the latter. ULTIMATEAUTOMIZER found three results that could not be identified by CPACHECKER, which in turn delivered answers on 13 tasks where ULTIMATEAUTOMIZER did not. Examining the configuration comparisons, we see that the predicate analysis runs faster than its value counterpart, with both returning the same amount of definite results.  $k$ -induction was the most successful configuration, finding two thirds more answers when compared to the other analyses. All configurations differ in run times only when tasks take considerably longer than a few minutes, where we observe that  $k$ -induction found answers faster in those cases.

On seven of the 208 tasks, the value analysis could not find a counterexample that was feasible, and after a few seconds it stopped the analysis when identifying multiple such spurious counterexamples with the message “Another infeasible counterexample found which could not be removed from the ARG”.

On 139 of the 208 tasks, the predicate analysis gave up due to an unsupported verification technique. Currently, MathSAT has no support for interpolation over floating point numbers<sup>6</sup>. Thus, CPACHECKER is not able to derive interpolants for most of the tasks, as floats are heavily used, and has to abandon the analysis completely.

The run times of CPACHECKER reveal that over 50% of the results were found in the

---

<sup>5</sup><https://github.com/diffblue/2ls/issues/123>

<sup>6</sup>[https://groups.google.com/d/msg/cpachecker-users/JBDRFPY2\\_II/e0a0veptBAAJ](https://groups.google.com/d/msg/cpachecker-users/JBDRFPY2_II/e0a0veptBAAJ)

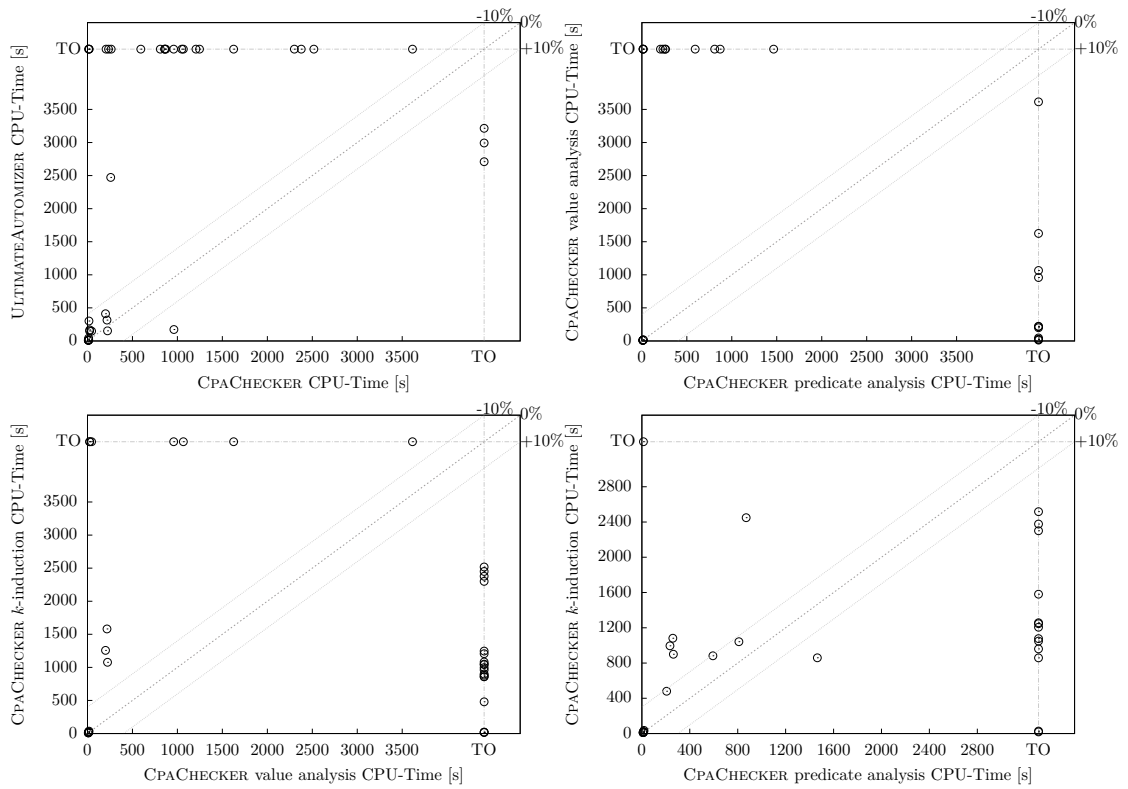


Figure 33: Comparison of CPU-times of CPACHECKER against ULTIMATEAUTOMIZER, and its configurations value analysis, predicate analysis and  $k$ -induction against each other.

first four minutes. Seven proofs were found after 20 and in no more than 60 minutes, as shown in Figure 34.

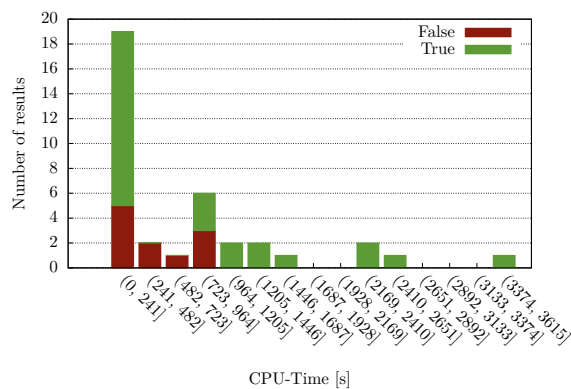


Figure 34: The CPU-times of CPACHECKER at which results were found.

Finally, we present an Euler’s diagram of the result coverage of the three different configurations in Figure 35. While predicate analysis was almost fully covered by the other two, value analysis and  $k$ -induction both found a considerable subset that was not identified by their counterparts.

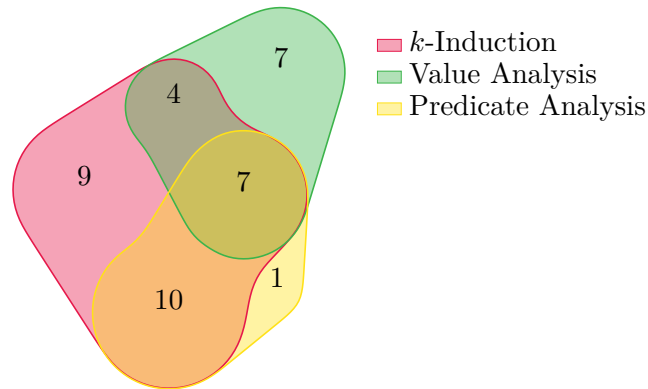


Figure 35: The Euler’s diagram comparing the result coverage of the three different configurations of CPACHECKER.

**Bugs and issues:** During the course of this work, we identified several issues in CPACHECKER. Firstly, C typedefs were not resolved correctly<sup>7</sup>. This bug initially prevented the tool from running on the case studies completely, although it was fixed quickly by the tool developers. After discovering the CBMC switch bug, we checked whether such behavior was also present in CPACHECKER. We found that switch-local variables are not represented internally at all, and thus ignored<sup>8</sup>. As we tried to run CPACHECKER with Z3 as its SMT-solver, we were deterred by a bug in the Z3-abstraction of JavaSMT<sup>9</sup>. Thus, we were only able to run CPACHECKER on its default SMT-solver MathSAT. Lastly, an exception was thrown on one DSR verification task for the  $k$ -induction configuration, where CPACHECKER seems to construct an invalid verification witness and identifies that issue, stating that “bad-state blocking invariants should be present if (and only if) induction failed”.

### 5.2.7 UltimateAutomizer

With only 19 definite answers, ULTIMATEAUTOMIZER was the second least successful verifier after SMACK. Although we observed mostly memory exceedance instead of time outs, most of them occurred in the last 15 minutes of run time. In the cases where ULTIMATEAUTOMIZER found a result, it mostly did so within seven minutes, as depicted in Figure 36. Three counterexamples and one proof were identified after half an hour.

<sup>7</sup><https://groups.google.com/forum/#!topic/cpachecker-users/wTqH0edB0b0>

<sup>8</sup>[https://groups.google.com/forum/#!topic/cpachecker-users/\\_bH55x\\_IN0w](https://groups.google.com/forum/#!topic/cpachecker-users/_bH55x_IN0w)

<sup>9</sup><https://groups.google.com/forum/#!topic/cpachecker-users/6wv6fgwHnk4>

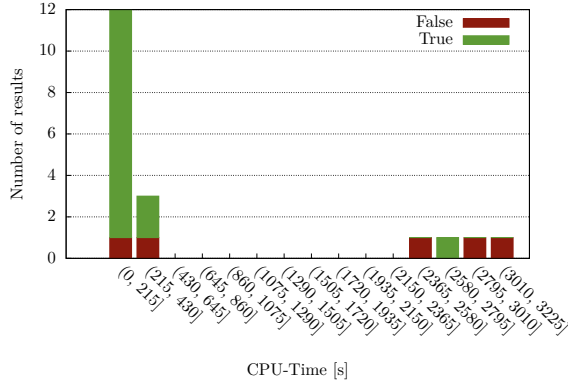


Figure 36: The CPU-times of ULTIMATEAUTOMIZER at which results were found.

**Bugs and issues:** ULTIMATEAUTOMIZER exhibits some erroneous behavior in both case studies: For ECC, it crashes on seven tasks when trying to compare two floating point variables (e.g. when constructing the following formula: `( _ FloatingPoint 8 24) ( _ FloatingPoint 8 24) Bool`)), although the exact circumstances are unclear. On DSR, we observed five tasks leading to a non-recoverable type mismatch with ULTIMATEAUTOMIZER reporting `Type mismatch (C_FLOAT != C_INT)` in an assignment statement.

### 5.2.8 k-Induction

Similar to CBMC, we report the depths at which K-INDUCTION found proofs. The counterexample depths are equal to those presented in Table 10, as we used CBMC as the back-end verifier. Table 11 states the number of proofs found at each depth. We again find that no proof needs a depth greater than three, and for DSR, every proof was found with  $k = 1$ .

Table 11: The depths at which the proofs were found by K-INDUCTION.

Depth	ECC		DSR		Total	
1	25	64%	24	100%	49	71%
2	14	36%	0	0%	14	35%
3	6	0%	0	0%	6	9%

Employing  $k$ -induction on top of CBMC increases the CPU-times of counterexample identification drastically, as depicted in Figure 37. In some cases, bugs were only identified after half an hour, whereas plain CBMC returned in a matter of minutes. We again observe that over half the answers could be given within five minutes. There were ten proofs that could only be found at around one hour of run time. We additionally report that four proofs were identified after the time limit of two hours, although we do not

include such answers in our data.

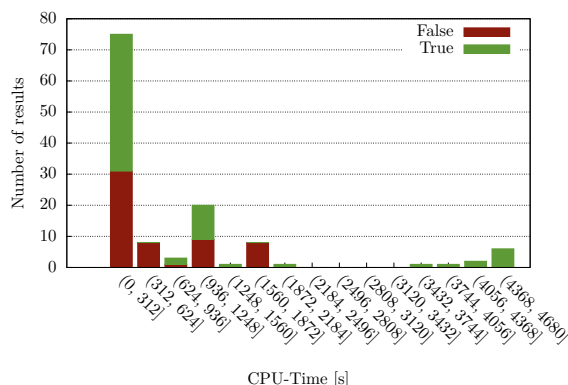


Figure 37: The CPU-times of K-INDUCTION at which results were found.

**Bugs and issues:** When using CBMC in version 5.10 for the induction step, run times on valid verification tasks increased threefold to about three to four hours. The presented results were gathered using the older version 5.8, where such behavior is not present.

### 5.3 Effect size of leverage points

#### 5.3.1 Code

Firstly, we present the effects of the applied code transformations: *Variable moving*, *Slicing*, and *Value analysis*. Table 12 gives a listing of the effect and significance value of each transformation for every verifier and configuration that was analyzed. Here, positive effects with regards to the verification goal – i.e. reduced time or increased result coverage – are marked in green, and negative effects in red, but only in case they were significant ( $p < 0.05$ ). Non-significant measurements are grayed out. Significance values were determined by using a paired sample  $t$ -test. The effect size is given in two ways: The relative difference of means, which is calculated by dividing the mean of the original by the mean of the effect measurements, and Cohen’s  $h$  respectively  $d$ , which gives a normalized indicator for the effect size.

Note that due to the bug of CBMC, as described in Section 5.2.1, we were not able to include the tool in this analysis. Further, we could not measure data on unsliced code for 2LS due to its expression simplification bug that prevented 2LS from running on the original code. This bug did not manifest itself in the slices, and thus only the effect from slicing to the value analysis was observed. For further details on the bug, we refer to Section 5.2.4.

Table 12: The significance and effect sizes of the analyzed code leverage points.

Verifier	Category	Metric	Moved	Sliced	Assumed	
<b>All verifiers</b>	Results	Rel. differ.	0.02	2.70	-0.08	
		Cohen's $h$	0.01	0.68	-0.06	
		Significance	1.000	0.000	0.005	
	CPU-Times	Rel. differ.	-0.58	-0.29	2.80	
		Cohen's $d$	-0.32	-0.13	0.50	
		Significance	0.049	0.225	0.000	
ESBMC	Results	Rel. differ.	0.00	0.24	-0.36	
		Cohen's $h$	0.00	0.20	-0.37	
		Significance	1.000	0.006	0.000	
	CPU-Times	Rel. differ.	-0.31	-0.19	2.40	
		Cohen's $d$	-0.13	-0.08	0.32	
		Significance	0.358	0.386	0.142	
	<i>Incremental</i>	Results	Rel. differ.	-0.08	0.18	-0.46
			Cohen's $h$	-0.05	0.09	-0.3
			Significance	1.000	0.625	0.109
CPU-Times		Rel. differ.	0.07	-0.40	0.07	
		Cohen's $d$	0.02	-0.19	0.03	
		Significance	0.243	0.321	0.357	
<i>k-Induction</i>	Results	Rel. differ.	0.03	0.27	-0.32	
		Cohen's $h$	0.04	0.35	-0.52	
		Significance	1.000	0.008	0.002	
	CPU-Times	Rel. differ.	-0.42	-0.13	2.40	
		Cohen's $d$	-0.18	-0.05	0.36	
		Significance	0.338	0.620	0.143	
2LS	Results	Rel. differ.			0.01	
		Cohen's $h$			0.02	
		Significance			1.000	
	CPU-Times	Rel. differ.			6.80	
		Cohen's $d$			0.84	
		Significance			0.000	
	<i>Incremental</i>	Results	Rel. differ.			0.00
			Cohen's $h$			0.00
			Significance			1.000
CPU-Times		Rel. differ.			7.00	
		Cohen's $d$			0.95	
		Significance			0.000	
<i>k-Induction</i>	Results	Rel. differ.			0.00	
		Cohen's $h$			0.00	

*Continued on next page*

Verifier	Category	Metric	Moved	Sliced	Assumed
<i>k-Invariant</i> <i>k-Induction</i>	CPU-Times	Significance			1.000
		Rel. differ.			6.70
		Cohen's <i>d</i>			0.95
	Results	Significance			0.000
		Rel. differ.			0.06
		Cohen's <i>h</i>			0.04
	CPU-Times	Significance			1.000
		Rel. differ.			3.00
		Cohen's <i>d</i>			1.30
		Significance			0.000
CPACHECKER	Results	Rel. differ.	-0.08	2.50	-0.08
		Cohen's <i>h</i>	-0.03	0.50	-0.05
		Significance	1.000	0.000	0.250
	CPU-Times	Rel. differ.	-0.79	-0.62	1.10
		Cohen's <i>d</i>	-0.76	-0.35	0.35
		Significance	0.083	0.455	0.048
	Results	Rel. differ.	0.00	0.62	-0.15
		Cohen's <i>h</i>	0.00	0.24	-0.09
		Significance	1.000	0.062	0.500
	CPU-Times	Rel. differ.	-0.97	1.80	-0.59
		Cohen's <i>d</i>	-0.99	0.42	-0.30
		Significance	0.084	0.410	0.197
	Results	Rel. differ.	-0.50	9.00	-0.10
		Cohen's <i>h</i>	-0.12	0.64	-0.05
		Significance	1.000	0.004	1.000
	CPU-Times	Rel. differ.	-0.04	-1.00	5.30
		Cohen's <i>d</i>	0.00	0.00	0.40
		Significance	1.000	1.000	0.346
Results	Rel. differ.	0.00	6.50	0.00	
	Cohen's <i>h</i>	0.00	0.75	0.00	
	Significance	1.000	0.000	1.000	
CPU-Times	Rel. differ.	-0.024	-0.96	1.80	
	Cohen's <i>d</i>	-0.56	-73.00	0.68	
	Significance	0.550	0.009	0.013	
ULTIMATEAUTOMIZER	Results	Rel. differ.	0.67	0.60	0.25
		Cohen's <i>h</i>	0.15	0.18	0.10
		Significance	0.500	0.508	0.500
	CPU-Times	Rel. differ.	0.01	-0.90	0.81
		Cohen's <i>d</i>	0.03	-32.00	0.33
		Significance	0.840	0.020	0.303

Overall, we report a significant medium-to-large effect for the result coverage of static slicing, although it neither changed verification times significantly nor effectively. Variable moving on the other hand leads to a significant medium-to-large reduction of verification times, but does not increase result coverage. Using value analysis for variable assuming increases verification times and reduces result coverage significantly. Those effects were present in most of the verifiers. Notable differences include CPACHECKER’s  $k$ -induction configuration and ULTIMATEAUTOMIZER, on which slicing lead to a significant CPU-time reduction of 96% respectively 90%.

Additionally, the direct effect of slicing on the code size – measured in *source lines of code* (SLOC) – is presented in Figure 38. Here, the SLOC of each sliced verification task is divided by the SLOC of the unsliced task, where both measures are taken from the Frama-C normalized AST code. The figure presents the resulting percentages of the ordered relative sizes.

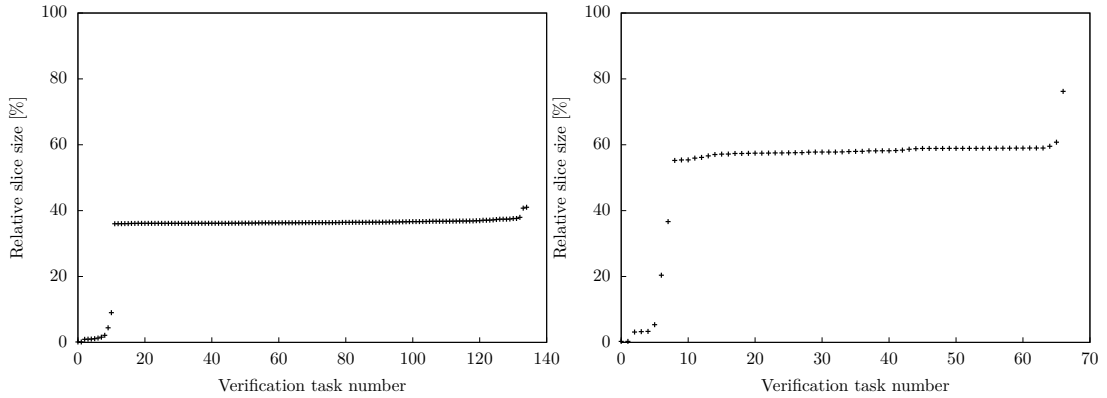


Figure 38: The relative sizes of the slices resulting from Frama-C, for ECC and DSR.

We report a mean reduction of the original SLOC to  $33\% \pm 10\%$  for ECC and  $52\% \pm 17\%$  for DSR. In total, 17 of the 208 tasks could be reduced to less than 10%. Of those, six tasks contained fewer than one percent of the original SLOC.

### 5.3.2 Verifiers

To firstly asses what role the back-end SMT-solvers take in software verification, we measured the time that was spent in the solvers, shown in Table 13. The measurements were taken for all verifiers that support some kind of SMT-solving time output. We measured the time for all verification tasks, even those on which no result was delivered. The CBMC-based tools spent around 30% to 40% of their verification time with SMT-solving. For CPACHECKER, this number increases to almost three quarters, whereas for

ULTIMATEAUTOMIZER it lies just around 2%. An increase in relative SMT-time was observed for both ESBMC and CBMC for the DSR case study.

Table 13: The mean and standard deviation of the relative amount of CPU-time that was spent in an SMT-solver.

Verifier	ECC	DSR	Total
CBMC	28.7% ± 23.9%	69.3% ± 29.7%	43.0% ± 32.5%
ESBMC	21.0% ± 27.1%	35.8% ± 14.5%	27.1% ± 23.8%
CPACHECKER	78.9% ± 36.4%	68.4% ± 38.0%	74.5% ± 37.8%
ULTIMATEAUTOMIZER	1.1% ± 0.7%	3.3% ± 4.2%	2.1% ± 3.1%
K-INDUCTION	36.9% ± 17.7%	34.2% ± 26.7%	36.1% ± 20.8%

In a comparable approach to the code transformation analysis, we collected the significance values and effect sizes of the SMT-solver configurations, depicted in Table 14. We measured the effect of the following configurations against the default: CBMC with Z3, MathSAT and CVC4, ESBMC with Boolector and ULTIMATEAUTOMIZER with Z3 and MathSAT.

It was found that using any SMT-solver for CBMC lead to no results, meaning that we only observed time outs after two hours. Thus, CBMC was excluded from the effect analysis, as there was nothing to observe.

For ESBMC, as stated previously, it was found that using Boolector lead to inconsistent results. Subsequently, we also excluded ESBMC from the effect analysis, as it did not offer another configuration to be tested.

Table 14: The significance and effect sizes of the analyzed verifier leverage points.

Verifier	Configuration	Solver	Measurement	Metric	Value	
ULTIMATE AUTOMIZER	<i>Default</i>	<i>WOLF</i>	<i>MathSAT</i>	Results	Rel. differ.	0.00
					Cohen's <i>h</i>	0.00
					Significance	1.000
				CPU-Times	Rel. differ.	0.49
					Cohen's <i>d</i>	0.21
					Significance	0.262
	<i>refinement</i>	<i>Z3</i>		Results	Rel. differ.	1.40
					Cohen's <i>h</i>	0.22
					Significance	0.001
				CPU-Times	Rel. differ.	0.56
					Cohen's <i>d</i>	0.20
					Significance	0.475

For ULTIMATEAUTOMIZER, we observed the same result coverage when using MathSAT

instead of the WOLF refinement. Further, mean run-times increased by 49%, although this effect was not significant ( $p = 0.262 \not\prec 0.05$ ). For Z3, with  $p = 0.001 < 0.05$  a significant result coverage effect was found, increasing the results by 140%. Cohen’s  $h$  indicates a medium sized effect. The CPU-times increased by 56%, but non-significantly ( $p = 0.475 \not\prec 0.05$ ).

## 5.4 Influence of the verification property

Section 5.1 states that some verification tasks are solvable by the selected verifiers, while others are not. This section presents analyses on the difference between the group of unverified and verified specifications.

Table 15: The percentages of the unverified tasks for their assessed categories.

Case Study	All	Initial Invariant	Global Invariant	Global Trigger-Response	Automaton Initial Condition	Automaton Transition
<i>ECC</i>	41.9%	40.0%	42.4%	00.0%	66.7%	48.1%
<i>DSR</i>	29.2%	50.0%	26.9%	66.7%	50.0%	26.5%

The share of unverifiable tasks with respect to the employed pattern is presented in Table 15. It additionally states the percentage of unverified automaton initial conditions and transitions. We find that for ECC, the share of unverified invariants matches the overall percentage with about 40%. For DSR on the other hand, we observe an increased relative amount of unverified initial invariants and trigger-response patterns. The verifiers were less successful on ECC’s automaton transitions, whereas on the automatons of the DSR case study we find less unverified tasks. Both case studies exhibit an increased share of unverified automaton initial conditions.

We also examine the significance of differences between the groups of verified and unverified specifications for various metrics. Those measurements are presented in detail in Section 3.1. For the analysis, we used a Mann-Whitney-U test where the groups were split into *verified* ( $n = 79$  for ECC and  $n = 51$  for DSR) and *non-verified* ( $n = 57$  for ECC and  $n = 21$  for DSR). The statistical test then checks for significant differences between the metrics of both groups. Table 16 reports the computed asymptotic significance values and the relative difference of the means with respect to the non-verified group. For example, a relative mean difference of +10% implies that the group of unverified tasks had an increase of ten percent for that measure.

We find that no significance falls below the standard threshold of  $\alpha = 0.05$ . For ECC, the significance of the number of global variables (mean difference of +1%) and the loop count in the slice (mean difference of -5%) leads to  $p < 0.2$ . For DSR, this is the case for the number of function calls in the slice (mean difference of +1%), the number of directly accessed floating point variables (mean difference of -44%) and the total number of indirectly accessed variables (mean difference of +26%).

Table 16: Significance values and relative mean differences of the two result groups.

Metric	ECC		DSR	
	Sig.	Mean diff.	Sig.	Mean diff.
<i>Specification lines of code</i>	0.742	-5%	0.624	+112%
<i>Slice lines of code</i>	0.327	+2%	0.409	±0%
<i>Slice decision points</i>	0.979	+2%	0.266	+2%
<i>Slice global variables</i>	0.182	+1%	0.330	-2%
<i>Slice if statements</i>	0.942	+1%	0.255	+3%
<i>Slice loops</i>	0.153	-5%	0.635	-2%
<i>Slice assignments</i>	0.368	+2%	0.691	-1%
<i>Slice function calls</i>	0.626	+2%	0.153	+1%
<i>Slice pointer dereferences</i>	0.975	±0%	0.406	-4%
<i>Slice cyclomatic complexity</i>	0.995	+2%	0.258	+2%
<i>Array access</i>	0.396	±0%	0.512	+300%
<i>Floating point comparisons</i>	0.407	+7%	0.330	+388%
<i>last usage</i>	0.358	-22%	0.501	-10%
<i>Variable direct access</i>	0.744	-4%	0.453	+6%
<i>Float direct access</i>	0.711	-5%	0.111	-44%
<i>Integer direct access</i>	0.727	-4%	0.318	+13%
<i>Variable indirect access</i>	0.438	+9%	0.117	+26%
<i>Float indirect access</i>	0.486	+9%	0.464	+18%
<i>Integer indirect access</i>	0.689	+7%	0.106	+29%

## 5.5 Comparison to BTC

To compare the verification results to the BTC verifier, we reflected the data of the 208 verification tasks back to the 107 original requirements. The answers of BTC were taken from [Berger et al. 2018]. Note that even though we present an attempt of comparison, this procedure does not depict a completely fair relation – the BTC results were specified separately by a different person, meaning that BTC did not run on the exact same tasks as the academic verifiers did. Thus, the hereafter presented results have to be considered under the fact that tasks may be interpreted and formalized differently. It is for that reason that we also do not present an Euler’s diagram as we did in Section 5.1.

Figure 39 and 40 show the results of each verifier, including BTC, on those reflections. We also present an “All” column representing the joined answers of all examined verifiers except BTC.

We observe that with the inclusion of the κ-INDUCTION tool and by combining all verifiers, we reach the same total coverage as BTC – around 74.6% versus 74.5%. Although BTC exhibits better performance in the ECC case study, the DSR case study favors the academic tools, with the biggest chunk of answers resulting from CBMC and the κ-INDUCTION tool.

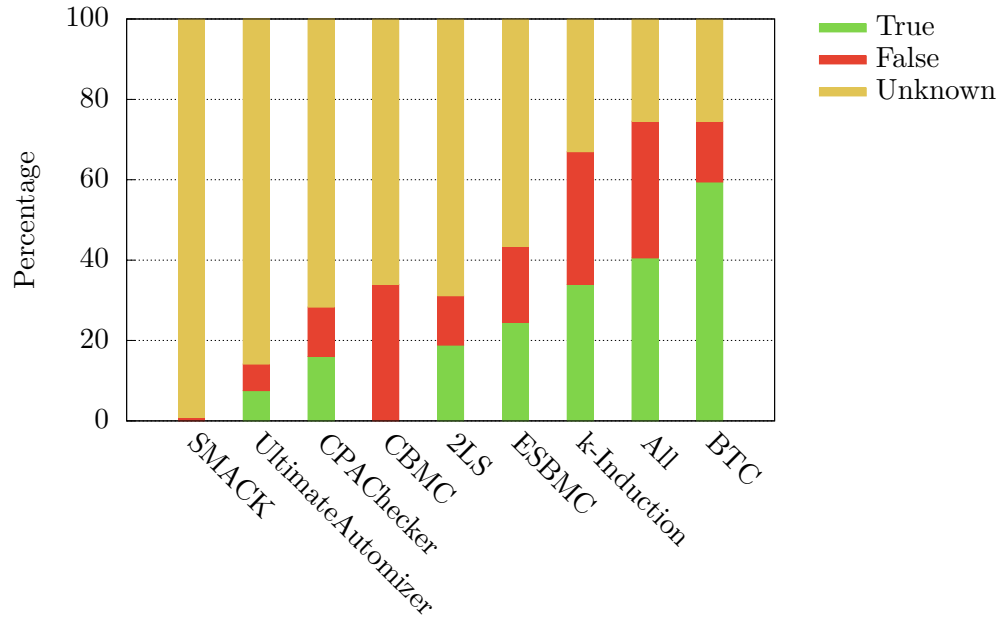


Figure 39: The overall results reflected back to the requirements, compared to BTC.

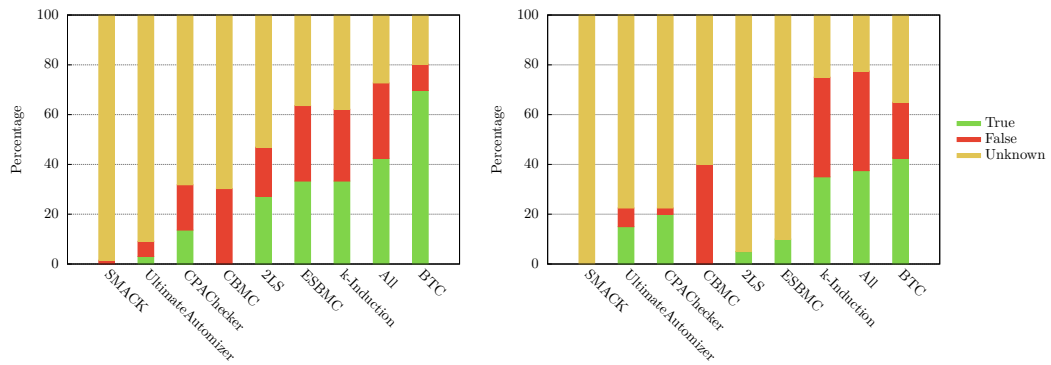


Figure 40: The results for each case study – ECC and DSR – reflected back to the actual requirements, compared to BTC.

## 6 Discussion

In this section, we present an interpretation of the benchmark data. We discuss the overall performance of academic software verifiers on our two case studies as well as their individual results. Further, we give a recommendation on which tools and leverage points are likely to be promising, and finally assess the effectiveness of the current academic implementations against an industrial verifier.

## 6.1 Overall results

Firstly, a combined coverage of 62.5% over all verifiers indicates that there is still room for improvement. As the competing verifiers are often optimized for improved results in the SV-COMP, a likely explanation lies in the type of its verification tasks. Those consist of only few real world applications and have very little coverage of industrial embedded-style programs. Considering this, integrating a number of industrial samples into the testing process has the possibility of greatly benefiting the verifier development.

$k$ -induction manifests itself as the most promising verification technique for the analyzed embedded-style programs. All verifiers that integrate such an approach out-perform their competition on our case studies. This effect likely arises due to the inherent structure of the verification task: a single loop over which the induction has to be performed, where the property is checked after each execution step. On the other hand,  $k$ -induction can be limited in power when moving away from showing loop invariants. In a plain version,  $k$ -induction is generally not able to find proofs for properties that are located at arbitrary points in the loop. An example of such a property is an invariant that has to hold in every program state – as opposed to only at the end of each loop iteration. The analysis of the functional requirements unveiled that they only contain safety properties that have to be checked after each step. Thus, exploiting this specific property pattern as well as having a bounded step in exactly one unbounded loop allows for an easy application of  $k$ -induction.

In case a  $k$ -induction verifier is not able to find a bound on the step function, no answer will be returned as the first induction step for  $k = 1$  does not terminate. This is possibly the case on the DSR verification tasks, which contain a bounded loop in one of the library functions. This loop is not recognized as being bounded by some verifiers, indicating that  $k$ -induction does not return once the loop is reached. For CBMC, this problem can be solved by setting the loop bounds manually – e.g. `-unwind 256` – but for most other verifiers, a fine-grained loop bound control was not available. This may be the reason of CPACHECKER’s and ULTIMATEAUTOMIZER’s improved performance on DSR compared to the  $k$ -induction tools. Furthermore, this can explain the observation of less overall result coverage on DSR, although it is by every measure the easier case study – it contains less global variables, floating point arithmetic, arrays, void pointers, source lines of code, et cetera.

Regarding the run times on the verification tasks, most results were returned early in the available time frame. Furthermore, bugs were found only at shallow depths – one to three – and  $k$ -induction proofs for low  $k$  values. If one uses such tools in an industrial setting, choosing a high run time for one verifier may not pay off. Instead, one should employ a variety of approaches and verifiers either parallelly or sequentially, as the Euler’s diagram of Figure 24 indicates that even tools of the same technique – e.g.  $k$ -induction – cover distinct areas of verification tasks. The return of time investment appears relatively low, implying that in case of a sequential approach, executing each verification tool for at most a couple of minutes can deliver almost all results. It has to be noted that some

tools – such as CPACHECKER, K-INDUCTION, and ULTIMATEAUTOMIZER, need a few seconds of initialization time. Thus, we advise to not restrict run times by too much.

In an industrial setting, it is advantageous to verify counterexamples and proofs in order to avoid false negatives and positives. Due to lack of time, we neither collected nor validated such witnesses. The amount of discovered bugs in some of the verifiers leads us to believe that there is a real possibility of false positives or negatives. Due to this, and based on the fact that ULTIMATEAUTOMIZER and CPACHECKER already support validation techniques, we strongly advise for the implementation of such a routine.

We find that no verifier uses exceptionally large amounts of memory, with both Java-based tools – CPACHECKER and ULTIMATEAUTOMIZER – exhibiting a higher consumption. If run time is restricted to less than an hour, software verification is even possible on a typical development machine. Such a local verification has the advantage of continuously and directly checking against property violations as new features are being implemented.

Observing that most results were found with little memory and in a short time frame leads to the conclusion that not resources are the limiting factor of software verification, but rather the theoretical strength of the implemented algorithms and techniques. Exemplary for this is the program presented in Figure 10, where  $k$ -induction is limited by its conceptual approach. Increasing computational power likely reduces run times, but we predict only a small effect on result coverage.

## 6.2 Verifier results

All verifiers exhibited bugs and issues at some point of the benchmarking process. We draw the conclusion that developing such a complex tool is inherently prone to mistakes. One of the most important features of a verifier is giving guarantees over its input program, and thus delivering reliable answers. If those results become unreliable, user’s trust in an application of formal methods may be reduced. It has been found that a sufficiently complex system requires appropriate trust [Lee and See 2004]. However, most examined verifiers are developed in an academic setting with limited resources and a focus on evaluating research approaches. While stability and reliability is still a key target, academic development processes often leave little room for the robustness often required in the industry. To illustrate this point, we examined how the verifiers run on the exemplary code of Figure 41.

All verifiers but SMACK prove the value of  $x$  being four, i.e. the absence of any bug. SMACK concludes that it is equal to three. In fact, the code exhibits undefined behavior and as such, any value of  $x$  can be assumed after line three, according to the C standard. If such a value is fixed, the verifier may miss out on bugs produced under certain compilers. For example, *gcc* version 8.2.1 sets  $x$  to four, whereas *clang* version 7.0.0 assumes  $x$  to be three.

This example visualizes the difficulty of implementing a program language semantics, es-

```

1 int main() {
2     int x = 0;
3     x = (x = 1) + (x = 2);
4     if (x != 4) __VERIFIER_error();
5     return 0;
6 }

```

Figure 41: An example program that exhibits undefined behavior by accessing `x` parallelly on line three.

pecially for C. In its core, any software verifier includes a complete parser and interpreter of its verification language, which poses a whole engineering challenge on its own. As an alternative approach, one could consider relying on a unifying semantics framework, such as the C implementation of the K framework [Roşu and Şerbănuţă 2010]. Note that, although it has been thoroughly tested, such a framework can not eliminate the occurrence of semantic errors completely.

### 6.2.1 CBMC

Due to its coverage of 27.9% of all tasks, we conclude that CBMC is the most powerful verifier for finding bugs on the examined studies. Neither ESBMC (4.8% coverage) nor 2LS (12.0% coverage) in their incremental settings were able to find nearly as much counterexamples, even though all bugs were located at depth three or shallower. The implemented fine-grained loop bound control was of particular usefulness in our case studies.

As 2LS is based on CBMC, it is to no surprise that we observe comparable run times. Considering this, ESBMC unexpectedly performed substantially slower compared to CBMC. Although also a fork, its performance may be attributed to the large amount of changes since 2008, as most of the internal framework was completely rewritten over time.

Guessing an a-priori loop bound for standard CBMC has shown to be difficult, as verification times vary between tasks. The incremental wrapper script proved its effectiveness by reducing run times and increasing result coverage through prevented time outs. We conclude that incremental bounded model checking is vital for verifying unbounded embedded-style programs.

### 6.2.2 CBMC Incremental

The findings indicate that CBMC INCREMENTAL is preferable over CBMC with an incremental wrapper script. Nevertheless, we recommend the latter, as the development of CBMC INCREMENTAL has stopped, meaning that bug fixes and performance improvements will not make their way into the incremental branch. It should also be considered that the pay-off is relatively small due to the absence of bugs deeper than three un-

rollings. We suspect that CBMC INCREMENTAL may show greater benefits if run on an unbounded loop that contains deeper bugs.

### 6.2.3 ESBMC

While  $k$ -induction has shown to be the most successful approach overall, we found significant differences between tools implementing this method. First off, all  $k$ -induction tools supplement each other, indicating that a combination will deliver a larger amount of results. Run times indicate that 2LS is preferable over ESBMC, but CPACHECKER is not, possibly due to ESBMC's advantage of specialization. Either K-INDUCTION or ESBMC finds proofs quickly, again pointing out that running such tools parallelly can be advantageous.

### 6.2.4 2LS

2LS offers a strengthened version of  $k$ -induction, called  $k$ -induction  $k$ -invariant. Here, the induction hypothesis is enhanced by abstract interpretation. As correct assumptions over the loop variables should not lead to less results overall, we did not expect a decrease in definite answers. Nevertheless, the observed behavior shows that plain  $k$ -induction is by far more successful. For this, we offer two explanations: Firstly, plain  $k$ -induction delivers false negatives, similar to the reported bug. Secondly, supplementary invariants increase the actual load on the verifier. This behavior was also observable when examining the leverage points, where adding `__VERIFIER_assume` statements with the results from static analysis increased CPU-times significantly and decreased result coverage, although only by a small amount. To determine the underlying issue, we used ULTIMATEAUTOMIZER to validate some of the correctness witnesses that were returned by  $k$ -induction but not by  $k$ -induction  $k$ -invariant. We did not find a false negative. Moreover, we did not identify any conflict of 2LS' answers to other verifiers. We conclude that most likely internal issues in 2LS' implementation of  $k$ -invariants hinders its performance.

Apart from its internal discrepancies, 2LS found results quickly and beats CPACHECKER's  $k$ -induction configuration by a wide margin. In comparison to K-INDUCTION and when disregarding timeouts for 2LS, we find a similar performance, most likely due to both verifiers being based on CBMC. In case that 2LS would have been able to run on DSR, we conjecture a similar performance to the K-INDUCTION tool.

### 6.2.5 SMACK

SMACK delivered the least amount of answers in this case study. Such behavior may be caused by the Mono-framework, which, compared to .NET under Windows, can lead to impediments in performance. Since we adhered to a fair competition, we did not run SMACK under Windows. The outcome could be different with .NET, but this

remains to be proven. The discovery of results way after the time limit was exceeded indicates performance issues. Furthermore, SMACK starts a large amount of processes. The necessary inter-process communication may be expensive and increases CPU-time.

### 6.2.6 CPAChecker

For CPACHECKER, a first evaluation of the data revealed the  $k$ -induction and value analysis configuration to be the most promising. Nevertheless, considering the efficacy of plain  $k$ -induction tools, CPACHECKER's version comes off both less effective and efficient. We again find that  $k$ -induction is impeded on the DSR case study. Similar to 2LS, CPACHECKER uses an enhanced approach. Counter-intuitively, we observe not as great of a performance as with plain  $k$ -induction. It is unclear whether this effect arises due to the strengthened invariants, or due to conceptual flaws in CPACHECKER. Indicative for the latter is the moderate performance of both other analyses.

When factoring in that the predicate analysis was able to run on only a third of all tasks, and delivered results on a quarter of those, we predict a better performance if CPACHECKER would have been able to interpolate over floats. To the point of writing, the researchers are implementing a different refinement scheme which may improve results for CPACHECKER in the future.

In contrast, the value analysis ran on most the tasks and delivered results on only 9% of them. Compared to the predicate analysis, we thus identify a limited relative performance. On the other hand, seven answers were returned that could not be found by the  $k$ -induction configuration. Due to this, we still recommend running the value analysis.

When comparing CPACHECKER to ULTIMATEAUTOMIZER, we find no substantial difference in run times. It has to be noted that CPACHECKER has the advantage of delivering answers on a superset of tasks when both tools are used in their default configuration. Due to this and the increased versatility of CPACHECKER, we recommend it over ULTIMATEAUTOMIZER for our use case.

For CPACHECKER, it may pay off to use longer run times, as some proofs were found between 20 and 60 minutes.

During the usage of CPACHECKER we found that tool development was very active and authors responded quickly to requests.

### 6.2.7 UltimateAutomizer

Typically, counterexamples were found early during our benchmarks. ULTIMATEAUTOMIZER on the contrary identified some counterexamples only after long run times. This can lead to the conclusion that underlying performance issues inhibit the verifier, as all bugs have a shallow depth. Most results were returned on only the smallest slices, which also acts as an additional indicator of such issues.

ULTIMATEAUTOMIZER is the youngest tool examined in this case study. Its relatively short development time may put its results into perspective: certain features that are essential to our case studies, such as array interpolation, were added as few as three years ago. Compared to the other tools, which can rely on a decade-long development of their frameworks – e.g. 14 years for CBMC and its forks as well as ten years for CPACHECKER – ULTIMATEAUTOMIZER has young roots. We conclude that there exists potential for improvement, as results in the SV-COMP are promising. Still, ULTIMATEAUTOMIZER seems not yet to be ready to tackle industrial-grade tasks.

### 6.2.8 *k*-Induction

The reasoning behind implementing *k*-induction on top of CBMC lies in the vast difference of identified bugs between CBMC and its competition of incremental bounded model checkers. After noticing promising results with other *k*-induction tools, we anticipated to lever the strength of such a matured bounded model checker to proof generation. Examining the result coverage, we conclude that K-INDUCTION is the most promising approach to verify the examined embedded programs.

Some proofs were found close to the time limit, most likely due to the *k*-induction step not being integrated into the verifier, but implemented on top. As we even found proofs after two hours, it may be beneficial to use longer run times for K-INDUCTION.

Besides harnessing the power of CBMC, K-INDUCTION's advantage lies in its ability to control loop bounds. Thus, the tool was able to perform *k*-induction on DSR, where other verifiers were not. Another reason for the improved result coverage is taking advantage of the case studies' specific form. Here, we run *k*-induction only on the main loop. Other verifiers try a combined induction step for all loops present, even if they are bounded. In such a combined induction step, each loop is unrolled *k* times, and the induction is performed over all unrollings. In basic terms, we incorporated the knowledge about the step function bound to derive improvements in the verification process.

Furthermore, and unlike 2LS, CBMC showed no bugs when running on the DSR verification tasks. This explains a large part of the margin between K-INDUCTION and 2LS.

As this tool implementation is merely a feasibility study, all results have to be taken with a grain of salt. Due to the limited time available, there is a high probability that it still contains bugs and may produce false negatives. On the other hand, the chance of false positives should be almost eliminated by using a well-tested back-end verifier, such as CBMC. A way to identify false proofs is the generation and external validation of verification witnesses. For time reasons, we were not able to implement such an additional feature into the tool, although it would greatly benefit its reliability.

## 6.3 Effect size of leverage points

### 6.3.1 Code

This study examined three possible leverage points on the input code. We identify static slicing as the most successful approach. When examining all results, it reduced verification times by 29%, although non-significantly. Its real strength shows itself in the result coverage: here, verifiers returned 270% more definite answers. This is most likely due to the fact that slicing removes unneeded parts of the program, that are, although reachable, not relevant to the satisfaction of the safety property. In some cases, slicing even reduces the program size drastically, as visualized in Figure 38. The discrepancy between increasing result coverage but having only a small effect on run times may be explained by internal optimizations of the verifiers. If they find a result, then those optimizations ensure that it does so in a timely fashion. On the other hand, if the code is too complex to be analyzable, slicing helps to remove structures on which the verifier may give up, such as complex arrays, dynamic memory allocations, or seemingly unbounded loops.

While the overall result coverage effect size of slicing is medium to large, not all verifiers and configurations exhibit this response. The greatest effect was observed for CPACHECKER’s predicate analysis and  $k$ -induction configurations, indicating that an integration of a slicing procedure can greatly benefit this tool. On the other hand, we found only a small effect for its value analysis. This can be explained by the applied abstractions, where a set of relevant variables is tracked and iteratively refined. This can mitigate the effect of slicing as both approaches reduce the observed program to relevant parts. The CPU-times of  $k$ -induction were reduced by 96% when slicing was applied. This leads us to believe that there are still performance optimizations to be added to CPACHECKER’s  $k$ -induction, as it also has not reached the efficacy of other  $k$ -induction tools.

For ESBMC, slicing leads to a medium effect, and is most effective on the  $k$ -induction setting. This is to no surprise, as symbolic execution in an incremental bounded model checker should only uncover program parts that are reachable.  $k$ -induction on the other hand can benefit from slicing, since program variables are non-deterministically set before each loop step. By decreasing the program’s state space, the verifier has to track less non-deterministic variables, which also reduces the load on the applied SMT-solver.

The usage of slicing shows only a small effect on the result coverage for ULTIMATEAUTOMIZER, although CPU times were drastically reduced. This may be another indicator of internal performance issues in ULTIMATEAUTOMIZER, as most other verifiers exhibited only a small effect on run times. It may be the case that abstractions are chosen too finely, and thus leading to an increased load on the verifier. By removing program parts, even fine abstractions can lead to quick results, as the abstraction’s state space is reduced.

Finally, it has to be noted that the effect of slicing may also be impacted by the nor-

malization of the code. Some verifiers may exhibit different efficiencies when running on Frama-C’s output versus the original code. This effect was not examined in this study, although it may contribute to improved results. For example, Frama-C moves some variables to their most-local block scope, such as the initialization block of switch statements.

Besides slicing, moving variables to their most local function scope has also shown to reduce verification times. We advise to apply this technique, but do not deem it to be an essential step in a successful verification tool chain. It does not reduce the actual state space, but allows for functions to be analyzed easier due to reduced interdependence. On CPACHECKER, we observe that variable moving reduces run times for unbounded model checking techniques, i.e. variable and predicate analysis. The effect on the variable analysis was close to significance, and exhibits a large size, indicating that for such unbounded model checking techniques, variable relocation may be more beneficial than for  $k$ -induction and bounded model checking.

Lastly, we applied static analysis to hint the verifiers with over-approximations of variable domains. This technique turned out to be unfavorable by increasing run times and decreasing result coverage. We hypothesize that adding a large amount of variable assumptions may increase the load on the verifier, as those assumptions have to be tracked. Not all variable domain restrictions may be useful in a successful analysis, and may interfere with abstraction techniques. Thus, we conclude that a more sophisticated approach is needed to integrate value analysis into a verification process. For example, one could identify variables whose domains are small and discrete, and inspect only those. This may reduce the amount of tracked assumptions while still delivering significant information to the verifier. Another reason for the reduced performance lies in the locality of the assumptions in the code: here, Frama-C is able to determine values at the end of each function. We may observe different results, especially for  $k$ -induction, if value domains are determined for the beginning and end of each loop iteration. Furthermore, our approach has the disadvantage of not being able to integrate into the verifiers. This means that we do not have information about the current state of abstractions, and for which variables hints may be beneficial. Thus, integrating static analysis into the verifiers directly, as CPACHECKER and 2LS do, can deliver better results. Such an approach also allows for a continuous refinement of variable domains during the verification process.

### 6.3.2 Verifiers

While most verifiers offer a variety of solver back-ends, we were only able to test the differences of SMT solvers for one verifier. Often, either interpolation support or issues in the usage of the solvers led to an inability to test them. As depicted in Table 13, solving times represent a large part of software verification. The standard deviations indicate a wide range of percentages, which further leads to the conclusion that solver involvement can be hard to predict. It has been shown that performance of solvers can vary between inputs, where heuristics of one solver may be advantageous on certain tasks but not on

others. Thus, not being able to evaluate the performance of the verifiers under different solvers likely prevents us from deriving improvements.

To further strengthen the previous point, we identified a positive impact when switching between solvers in `ULTIMATEAUTOMIZER`. Here, using `Z3` instead of a dynamic solver from the `WOLF` refinement strategy leads to an increase of 140% in result coverage. It is only due to this effect that `ULTIMATEAUTOMIZER` is competitive in this case studies. We did not find a significant difference when using `MathSAT`. This has to be expected, as an evaluation of the `WOLF` refinement strategy reveals that it uses `MathSAT` most of the time on our verification tasks.

On a side note, the relative time spent in a solver is drastically lower for `ULTIMATEAUTOMIZER` than for the other verifiers. This can be ascribed to possibly not measuring the full time spent in the solver, as `ULTIMATEAUTOMIZER` did not offer documentation on their statistics output. On the other hand, the verifier spends large amounts of time doing expensive automaton operations, such as cross products. Due to this, it may seem counter-intuitive that `ULTIMATEAUTOMIZER` is influenced by selecting `Z3` over the `WOLF` refinement. In fact, the verifier greatly relies on the quality of the interpolants. Thus, `Z3` most likely delivered more information through its interpolants, allowing `ULTIMATEAUTOMIZER` to exclude a larger and more relevant set of spurious counterexamples.

## 6.4 Influence of the verification property

To analyze which parts of the verification tasks correlate with the solvers' inability to return definite answers, we first looked at the distribution of unverified tasks among the property categories. Surprisingly, we found that initial invariants are often harder to verify than their global counterparts. A possible explanation involves the general complexity of the code, indicating that verifiability highly depends on the semantics of the property, as depicted in Figure 10 for  $k$ -induction. The same reasoning applies to the verification of automaton initial conditions, where we observed a similar pattern. In total, automaton transitions appear a bit harder on the verifiers. A reason for this may be that certain approaches, such as plain  $k$ -induction, are not able to derive bounds on the state space variables.

We observe that in total, two out of four trigger response tasks were solved. As  $n = 4$ , we can not draw a reliable conclusion here, albeit from stating that at least some trigger-response patterns are practically verifiable.

To identify challenges that are posed to the verifiers, we searched for distinct features in the group of unverified specification tasks. Interestingly, we could not find any meaningful differences between both groups. Conversely, some metrics were increased for the verified group, such as the direct variable access for ECC. We were able to identify two respectively three metrics for ECC and DSR which resulted in a significance of  $0.1 < p < 0.2$ . Such  $p$ -values can be used as identifiers on which measures to examine

further, although they most likely do not indicate reliable results. We conclude that for ECC, the global variables and loops in each slice may play a role in verifiability. For DSR, this may be true for direct floating point and indirect total variable access of the properties, although the first one exhibits the opposite relation: unverified tasks contain 44% less direct variable access. This strengthens the hypothesis that the collected metrics did not contain enough expressive power to reason about verifiability.

We conclude that identification of verifiability reasons is a non-trivial task and can most likely not be done with static and shallow semantic metrics. On the contrary, it seems that rather than the task’s syntax, semantics is the underlying reason. This reasoning is further justified by the lower verification probability of initial invariants.

## 6.5 Comparison to BTC

In their current state, academic verifiers are not able to overtake BTC for the examined industrial application. This is to no surprise, as BTC states to apply incremental bounded model checking and  $k$ -induction techniques on top of CBMC since version 4.3 [Schrammel et al. 2015]. Our feasibility study has also shown comparable result coverage for this exact combination. Furthermore, BTC is able to use “a comprehensive experimental evaluation over a large set of industrial embedded benchmarks” [Schrammel et al. 2015]. As discussed previously, most examined verifiers are optimized for academic examples rather than real-world applications, and having access to such a large test suite gives BTC the advantage of multiple years of specialization.

Although it appears that BTC was able to generate more proofs and less counterexamples, those differences most likely arise due to variations in requirement formalizations. Thus, we can not conclude that BTC is inherently weaker in producing counterexamples and stronger in providing proofs. On the contrary, a similar bug coverage compared to CBMC has to be expected, as it is used as a back-end in BTC. When specifying tasks for BTC, various property changes that were reflected in the code but not in the requirements were elaborated with the Ford team. For this thesis, we made an effort to keep the specifications as close to the requirements as possible. For example, an automaton transition contained a condition of the form  $x < y$  in its requirement. After examination, it was found that  $x \leq y$  was actually implemented, and the requirement was changed in the BTC formalization. Such discrepancies in the verification tasks can explain the offset of identified counterexamples between BTC and the academic verifiers.

Similar to our approach, BTC also implements static analysis techniques prior to their verifier usage. Here, “after some preprocessing like source-level slicing and internal-loop unwinding the resulting reachability task is given to CBMC” [Schrammel et al. 2017]. Thus, BTC may infer better results from there, as Frama-C’s slicing could not be applied to CBMC in this work.

Lastly, BTC has experimented with an incremental SMT-solver back-end, which is a natural fit for the application of incremental bounded model checking. They found that

it both increased result coverage and reduced run times on their industrial test suite [Neubauer et al. 2016].

Combining those advantages, it should be expected that BTC outperforms its academic competition. It has to be noted that some of the developers of the most successful tools in our case study, 2LS and CBMC, are conducting research in collaboration with BTC.

## 7 Future work

To refine the presented benchmarks of this study, an incorporation of the external validation of verification witnesses is crucial. Such an approach makes the presented results more robust and even allows for a more precise scoring system, similar to the SV-COMP. For example, one could give penalties for false answers and rewards for delivering correct witnesses.

Although we tried to include a wide variety of modern academic verifiers, we were not able to run an exhaustive test suite for all available tools. Specifically, SYMBIOTIC [Chalupa et al. 2017] has not been examined, although it has a chance of delivering competitive results to CBMC due to its symbolic execution approach. Furthermore, DEPTHK, a  $k$ -induction tool on top of ESBMC, was not included as it did not participate in the overall category of 2018’s SV-COMP, and especially not in the software systems sub-category [Rocha et al. 2017]. This indicates that in its current state of early development – being firstly presented in 2017 – it has not yet matured enough. By adding over-approximating loop invariants, it implements a promising approach to  $k$ -induction, as this may reduce verification times if the over-approximation is fine enough to deliver a proof. Thus, we advise to re-examine DEPTHK’s development status after some SV-COMP iterations.

As it was observed that static slicing on program level is a viable technique, it is advisable to continue investigation into other static analysis approaches. For example, one can adapt ideas from the BTC EmbeddedValidator and implement a static loop unwinding before verification. If a loop bound can be determined statically, the observed problems on the DSR case study, where some verifiers were not able to infer a bound on one of the library loops, could be diminished. It should be noted that we also identified some static analysis techniques as counter-productive, implying that such approaches should be carefully tested for effectiveness.

As we identified  $k$ -induction with CBMC as the most effective method of verifying the tasks at hand, we advise further development in that area. Future work for this tool may include a strengthened induction hypothesis approach, in a way that similar verifiers, e.g. CPACHECKER [Beyer et al. 2015] and 2LS [Brain et al. 2015], already do. Additionally and similar to DEPTHK, one can also rely upon over-approximating invariants, although one would need to incorporate external proof validation to ensure the correctness of the over-approximation. The most urging step in the development of K-INDUCTION is the output of verification witnesses. Without those, the verifier can not deliver the necessary reliability. Furthermore, the tool is specialized to one-loop programs with one invariant being checked exactly after each main loop step. To be able to broaden the range of input programs, one can rely on techniques to elevate one-loop  $k$ -induction to (nested) multiple loops [Gadelha et al. 2017]. Another shortcoming of K-INDUCTION lies in its missing implementation of a forward case, as this was not necessary for the applied use case. On other tasks, this may be required. Finally, there are multiple research advances in the area of  $k$ -induction. Implementing those can be a viable strategy to increase result

coverage in an industrial setting. For example, using loop accelerators may result in a more useful invariant more quickly [Madhukar et al. 2016]. There are also techniques to reduce the multi-process approach to a single verification task and thus avoiding the need for a separate base and induction step [Donaldson et al. 2011]. As the efficacy of `K-INDUCTION` is highly dependent on its back-end verifier, every improvement there has a high chance of direct translation. Thus, having an up-to-date incremental version of `CBMC` available will benefit `K-INDUCTION` greatly. Moreover, if such an incremental version would rely on an incremental solver, more performance improvements are to be expected [Neubauer et al. 2016]. We did not evaluate `K-INDUCTION` on different back-end verifiers, as `CBMC` has shown to be the most effective verifier for our case studies. It remains to be examined whether this is true for other use cases, or if some other back-end has the potential to produce better results. As our tool offers support for a configurable exchange of the back-end, such an examination can be performed with ease.

Apart from pursuing  $k$ -induction in further research, there exists a chance that the predicate analysis of `CPACHECKER` may deliver better results in a future version. Once the floating point interpolation scheme has been added to `CPACHECKER`, a re-evaluation is advisable.

Finally, the data showed that distinguishing unverifiable from verifiable tasks is a hard endeavor. Syntactic and shallow semantic metrics did not predict verification success. Thus, it is likely that the underlying semantics of the code and property play a significant role. In practice, it may be beneficial to the end-user to obtain an a-priori estimate of the chance of a successful verification. Furthermore, predicting which tool or technique has the best return of time investment leads to a more efficient application. In total, such an approach can deliver an increased trust in formal methods, as long as the predictions are reliable enough. For this, a machine learning approach may be considered, as presented in [Tulsian et al. 2014], although more data on more case studies will be needed. In our case,  $n = 2$  is too small of a sample size to apply the described methods.

## 8 Conclusion

In this work, we evaluated the current state of academic verification tools with regard to their applicability to two challenging case studies in the area of automotive embedded C code.

We firstly transformed the given functional requirements to formal specifications and created a set of 208 verification tasks to evaluate.

We selected a promising subset of verifiers competing in the SV-COMP. They implement a multitude of approaches, ranging from bounded model checking to abstraction techniques. Prior to verification, we applied various code transformations and verifier configurations with the goal to enhance result coverage and run times.

$k$ -induction emerged to be the most successful approach in our use case, although efficacy varies significantly between implementations. CBMC was outstanding in identifying bugs. A custom implementation of  $k$ -induction on top of CBMC lead to the best coverage, although it currently does not support verification witnesses. Other approaches, such as predicate and trace abstractions, returned little results. Most verifiers either gave answers in a matter of minutes, or did not return results at all.

Static slicing has shown to be an effective way of increasing result coverage. It may be beneficial to apply various static analysis techniques a-priori, although success is not guaranteed. We observed that logic solvers carry out a major part of the verification work, and experimenting with replacing back-ends can be advantageous.

Finally, we found that verification success on such challenging tasks can not be easily predicted by syntactic and shallow semantic measures.

To summarize, based on the findings of this study, we currently suggest the following procedure to verify automotive embedded C programs:

- Use (strengthened)  $k$ -induction tools.
- Particularly, use an incremental CBMC version, enhanced with  $k$ -induction.
- Employ a variety of verifiers sequentially or parallely, restricted to a couple of minutes of run time each.
- Apply static slicing beforehand.
- Validate verification witnesses.
- Test academic verifiers on real-world industrial examples.
- Experiment with different back-end solvers, if possible.

## References

- Alan Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72. MIT Press, 1948. 1
- Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967. 1
- C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, oct 1969. ISSN 00010782. doi:10.1145/363235.363259. URL <http://portal.acm.org/citation.cfm?doid=363235.363259>. 1
- E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 85 LNCS, pages 169–181, 1980. ISBN 9783540100034. doi:10.1007/3-540-10003-2\_69. URL [http://link.springer.com/10.1007/3-540-10003-2\\_69](http://link.springer.com/10.1007/3-540-10003-2_69). 1
- J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 137 LNCS, pages 337–351, 1982. ISBN 9783540114949. doi:10.1007/3-540-11494-7\_22. URL [http://link.springer.com/10.1007/3-540-11494-7\\_22](http://link.springer.com/10.1007/3-540-11494-7_22). 1
- Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, jan 2003. ISSN 00045411. doi:10.1145/602382.602403. URL <http://portal.acm.org/citation.cfm?doid=602382.602403>. 2
- Ph Lacan, Jean Noel Monfort, L V Q Ribal, A Deutsch, and G Gonthier. Ariane 5 - The software reliability verification process. In *DASIA 98-Data Systems in Aerospace*, volume 422, page 201, 1998. 2
- Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004. 2
- Jean Souyris. Industrial experience of abstract interpretation-based static analyzers. In *Building the Information Society*, pages 393–400. Springer, 2004. 2
- Alexey Khoroshilov, Vadim Mutilin, Alexander Petrenko, and Vladimir Zakharov. Establishing linux driver verification process. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5947 LNCS, pages 165–176, 2010. ISBN 3642114857. doi:10.1007/978-3-642-11486-1\_14. URL [http://link.springer.com/10.1007/978-3-642-11486-1\\_14](http://link.springer.com/10.1007/978-3-642-11486-1_14). 2

- Steve McConnell. Code Complete: A Practical Handbook of Software Construction. *Design*, 9(3):919, 2004. ISSN 1477-0539. doi:10.1039/c1ob90002a. URL <http://xlink.rsc.org/?DOI=c1ob90002a>. 2
- Barry W. Boehm and Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988. ISSN 00985589. doi:10.1109/32.6191. URL <http://ieeexplore.ieee.org/document/6191/>. 3
- Gerard J Holzmann. Economics of software verification. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 80–89. ACM, 2001. 3
- Daniel Dvorak. NASA Study on Flight Software Complexity. In *AIAA Infotech@Aerospace Conference*, 2009. ISBN 978-1-60086-979-2. doi:10.2514/6.2009-1882. 3
- Dirk Beyer. Software verification with validation of results (report on SV-COMP 2017). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10206 LNCS, pages 331–349, 2017. ISBN 9783662545799. doi:10.1007/978-3-662-54580-5\_20. URL [http://link.springer.com/10.1007/978-3-662-54580-5\\_{\\_}20](http://link.springer.com/10.1007/978-3-662-54580-5_{_}20). 4, 53
- Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Successful use of incremental BMC in the automotive industry. In Stefan Leue and Pedro Merino, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9128 of *Lecture Notes in Computer Science*, pages 62–77, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 9783319194578. doi:10.1007/978-3-319-19458-5\_5. URL [http://uk.mathworks.com/products/slidesignverifierhttp://link.springer.com/10.1007/978-3-540-79707-4http://www.springerlink.com/index/10.1007/978-3-540-79707-4http://link.springer.com/10.1007/978-3-319-19458-5\\_{\\_}5](http://uk.mathworks.com/products/slidesignverifierhttp://link.springer.com/10.1007/978-3-540-79707-4http://www.springerlink.com/index/10.1007/978-3-540-79707-4http://link.springer.com/10.1007/978-3-319-19458-5_{_}5). 4, 33, 83
- Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 29(5):911–931, sep 2017. ISSN 1433299X. doi:10.1007/s00165-017-0419-1. URL <http://link.springer.com/10.1007/s00165-017-0419-1>. 4, 83
- Bastian Schlich and Stefan Kowalewski. Model checking C source code for embedded systems. In *International Journal on Software Tools for Technology Transfer*, volume 11, pages 187–202, jul 2009. doi:10.1007/s10009-009-0106-5. URL <http://link.springer.com/10.1007/s10009-009-0106-5>. 4
- Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian

- rover software. In *Formal Methods in System Design*, volume 25, pages 167–198, sep 2004. doi:10.1023/B:FORM.0000040027.28662.a4. URL <http://link.springer.com/10.1023/B:FORM.0000040027.28662.a4>. 4
- Borja Fernández Adiego, Dániel Darvas, Jean Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. Bringing automated model checking to plc program development - A CERN case study. In *Advances in the Astronautical Sciences*, volume 12, pages 394–399, 2014. ISBN 9783902823618. doi:10.3182/20140514-3-FR-4046.00051. URL <https://linkinghub.elsevier.com/retrieve/pii/S1474667015374334>. 4
- Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O’Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 3–11. 2015. ISBN 9783319175232. doi:10.1007/978-3-319-17524-9\_1. URL [http://link.springer.com/10.1007/978-3-319-17524-9\\_{\\_}1](http://link.springer.com/10.1007/978-3-319-17524-9_{_}1). 4
- J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards verified automotive software. *ACM SIGSOFT Software Engineering Notes*, 30(4):1, jul 2005. ISSN 01635948. doi:10.1145/1082983.1083199. URL <http://portal.acm.org/citation.cfm?doid=1082983.1083199>. 4
- R. Venkatesh, Ulka Shrotri, Priyanka Darke, and Prasad Bokil. Test generation for large automotive models. In *2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings*, pages 662–667. IEEE, mar 2012. ISBN 9781467303422. doi:10.1109/ICIT.2012.6210014. URL <http://ieeexplore.ieee.org/document/6210014/>. 4, 6
- Johanna Nellen, Thomas Rambow, Md Tawhid Bin Waez, Erika Ábrahám, and Joost-Pieter Katoen. Formal Verification of Automotive Simulink Controller Models: Empirical Technical Challenges, Evaluation and Recommendations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10951 LNCS, pages 382–398. 2018. ISBN 9783319955810. doi:10.1007/978-3-319-95582-7\_23. URL [http://link.springer.com/10.1007/978-3-319-95582-7\\_{\\_}23](http://link.springer.com/10.1007/978-3-319-95582-7_{_}23). 4, 6, 9
- Matthew Bennion and Ibrahim Habli. A candid industrial evaluation of formal software verification using model checking. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 175–184, New York, New York, USA, 2014. ACM Press. ISBN 9781450327688. doi:10.1145/2591062.2591184. URL <http://dl.acm.org/citation.cfm?doid=2591062.2591184>. 4
- Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. Verifying Auto-generated C Code from Simulink. In *Formal Meth-*

- ods, pages 312–328, 2018. ISBN 978-3-319-95582-7. doi:10.1007/978-3-319-95582-7\_18. URL [http://link.springer.com/10.1007/978-3-319-95582-7\\_18](http://link.springer.com/10.1007/978-3-319-95582-7_18). 4, 9, 14, 72
- Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *10th European software engineering conference and 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'05)*, 30(5):263, 2005. ISSN 01635948. doi:10.1145/1081706.1081750. URL <http://osl.cs.uiuc.edu/http://portal.acm.org/citation.cfm?doid=1081706.1081750>. 4
- Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE - Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008. ISSN <null>. doi:10.1.1.142.9494. 4
- Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking: A comparative evaluation of the state of the art. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10629 LNCS, pages 99–114, 2017. ISBN 9783319703886. doi:10.1007/978-3-319-70389-3\_7. URL [http://link.springer.com/10.1007/978-3-319-70389-3\\_7](http://link.springer.com/10.1007/978-3-319-70389-3_7). 4
- Martin Bösch, Ralf Bogusch, Anabel Fraga, and Christian Rudat. Bridging the gap between natural language requirements and formal specifications. In *CEUR Workshop Proceedings*, 2016. doi:10.1007/s00761-013-2452-x. 4
- M G Ilieva and Olga Ormandjieva. Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation. In *LNCS*, volume 3513, pages 392–397. 2005. ISBN 3-540-26031-5, 978-3-540-26031-8. doi:10.1007/11428817\_45. URL [http://www.nlping.com/Articles/35130392\\_{\\_}NLDB\\_{\\_}05.pdf{\\_%}0Ahttps://pdfs.semanticscholar.org/2f78/5c33bc8ed5bf45a690e28fbc69dfef2f88f8.pdfhttp://link.springer.com/10.1007/11428817\\_{\\_}45](http://www.nlping.com/Articles/35130392_{_}NLDB_{_}05.pdf{_%}0Ahttps://pdfs.semanticscholar.org/2f78/5c33bc8ed5bf45a690e28fbc69dfef2f88f8.pdfhttp://link.springer.com/10.1007/11428817_{_}45). 4
- P. Chomicz, A. Müller-Lerwe, G.-P. Wegner, R. Busch, and S. Kowalewski. Towards the use of controlled natural languages in hazard analysis and risk assessment. In *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft für Informatik (GI)*, 2017. ISBN 9783885796633. 4
- Williame Rocha, Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Bernd Fischer. DepthK: A k-induction verifier based on invariant inference for C programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10206 LNCS, pages 360–364, 2017. ISBN 9783662545799. doi:10.1007/978-3-662-54580-5\_23. URL [http://link.springer.com/10.1007/978-3-662-54580-5\\_23](http://link.springer.com/10.1007/978-3-662-54580-5_23). 4, 85

- Jon Friedman. MATLAB/Simulink for Automotive Systems Design. In *Proceedings of the Design Automation & Test in Europe Conference*, pages 1–2. IEEE, 2006. ISBN 3-9810801-1-4. doi:10.1109/DATE.2006.243988. URL <http://ieeexplore.ieee.org/document/1656852/>. 5
- Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In *International Conference on Object-Oriented Information Systems*, pages 298–311. Springer, 2002. doi:10.1007/3-540-46105-1\_34. URL [http://link.springer.com/10.1007/3-540-46105-1\\_34](http://link.springer.com/10.1007/3-540-46105-1_34). 6
- Simulink design verifier. *Aircraft Engineering and Aerospace Technology*, 79(6):41–46, oct 2007. ISSN 00022667. doi:10.1108/aeat.2007.12779fad.006. URL <http://www.emeraldinsight.com/doi/10.1108/aeat.2007.12779fad.006>. 6
- Florian Leitner and Stefan Leue. Simulink Design Verifier vs. SPIN a Comparative Case Study. *Proceedings of FMICS*, 2008. doi:10.1.1.139.2119. URL [https://kops.ub.uni-konstanz.de/xmlui/bitstream/handle/urn:nbn:de:bsz:352-212922/Leitner-Fischer\\_212922.pdf?sequence=3](https://kops.ub.uni-konstanz.de/xmlui/bitstream/handle/urn:nbn:de:bsz:352-212922/Leitner-Fischer_212922.pdf?sequence=3). 6
- Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. ISSN 00985589. doi:10.1109/TSE.1976.233837. 8
- Maurice H Halstead and Others. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, 1977. 8
- Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*, pages 151–158, New York, New York, USA, 1971. ACM Press. ISBN 2090-2255 (Electronic). doi:10.1145/800157.805047. URL <http://portal.acm.org/citation.cfm?doid=800157.805047>. 16
- Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999a. doi:10.1007/3-540-49059-0\_14. URL [http://link.springer.com/10.1007/3-540-49059-0\\_14](http://link.springer.com/10.1007/3-540-49059-0_14). 18
- Armin Biere, Alessandro Cimatti, Edmund Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE conference on Design automation conference - DAC '99*, pages 317–320. IEEE, 1999b. ISBN 1-58113-092-9. doi:10.1109/DAC.1999.781333. URL <http://ieeexplore.ieee.org/document/781333/http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=781333%5Cnhttp://portal.acm.org/citation.cfm?doid=309847.309942>. 19
- Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded Model Checking and Induction: From Refutation to Verification. *15th International Conference, CAV 2003*,

- pages 14–26, 2003. ISSN 03029743. doi:10.1007/978-3-540-45069-6\_2. URL [http://link.springer.com/10.1007/978-3-540-45069-6\\_{\\_}2](http://link.springer.com/10.1007/978-3-540-45069-6_{_}2). 20
- Mikhail Y.R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *International Journal on Software Tools for Technology Transfer*, 19(1):97–114, feb 2017. ISSN 14332787. doi:10.1007/s10009-015-0407-9. URL <http://link.springer.com/10.1007/s10009-015-0407-9>. 20, 85
- Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto. Model checking embedded C software using k-induction and invariants. In *Proceedings - 2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015*, pages 90–95, sep 2016. ISBN 9781509001828. doi:10.1109/SBESC.2015.24. URL <http://arxiv.org/abs/1509.02471>. 22
- Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6887 LNCS, pages 351–368, 2011. ISBN 9783642237010. doi:10.1007/978-3-642-23702-7\_26. URL [http://link.springer.com/10.1007/978-3-642-23702-7\\_{\\_}26](http://link.springer.com/10.1007/978-3-642-23702-7_{_}26). 22, 86
- Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9206, pages 622–640, 2015. ISBN 9783319216898. doi:10.1007/978-3-319-21690-4\_42. URL [http://link.springer.com/10.1007/978-3-319-21690-4\\_{\\_}42](http://link.springer.com/10.1007/978-3-319-21690-4_{_}42). 23, 37, 85
- Kenneth L. McMillan. Symbolic Model Checking. In *Symbolic Model Checking*, pages 25–60. Springer US, Boston, MA, 1993. ISBN 978-0-7923-9380-1. doi:10.1007/978-1-4615-3190-6\_3. URL [http://link.springer.com/10.1007/978-1-4615-3190-6\\_{\\_}3](http://link.springer.com/10.1007/978-1-4615-3190-6_{_}3). 23
- Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7793 LNCS, pages 146–162, 2013. ISBN 9783642370564. doi:10.1007/978-3-642-37057-1\_11. URL [http://link.springer.com/10.1007/978-3-642-37057-1\\_{\\_}11](http://link.springer.com/10.1007/978-3-642-37057-1_{_}11). 24, 37
- Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, sep 1994. ISSN 01640925. doi:10.1145/186025.186051. URL <http://portal.acm.org/citation.cfm?doid=186025.186051>. 25

- Edmund Clarke. Counterexample-guided abstraction refinement. In *10th International Symposium on Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings.*, volume 2003-Janua, pages 7–8. IEEE Comput. Soc, 2003. ISBN 0-7695-1912-1. doi:10.1109/TIME.2003.1214874. URL <http://ieeexplore.ieee.org/document/1214874/>. 26
- Thomas a Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with BLAST. In *Model Checking Software*, pages 235–239. 2003. ISBN 3-540-40117-2. doi:10.1007/3-540-44829-2\_17. URL [http://link.springer.com/10.1007/3-540-44829-2\\_{ }17](http://link.springer.com/10.1007/3-540-44829-2_{ }17). 26
- Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997. doi:10.1007/3-540-63166-6\_10. URL [http://link.springer.com/10.1007/3-540-63166-6\\_{ }10](http://link.springer.com/10.1007/3-540-63166-6_{ }10). 27
- Kenneth L McMillan. Applications of Craig Interpolants in Model Checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005. doi:10.1007/978-3-540-31980-1\_1. URL [http://link.springer.com/10.1007/978-3-540-31980-1\\_{ }1](http://link.springer.com/10.1007/978-3-540-31980-1_{ }1). 27
- William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, sep 1957. ISSN 0022-4812. doi:10.2307/2963594. URL [https://www.cambridge.org/core/product/identifier/S0022481200069796/type/journal\\_{ }article](https://www.cambridge.org/core/product/identifier/S0022481200069796/type/journal_{ }article). 27
- Kenneth L McMillan. Lazy Abstraction with Interpolants. In *International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006. doi:10.1007/11817963\_14. URL [http://link.springer.com/10.1007/11817963\\_{ }14](http://link.springer.com/10.1007/11817963_{ }14). 28
- Ranjit Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3920 LNCS, pages 459–473. 2006. ISBN 3540330569. doi:10.1007/11691372\_33. URL [http://link.springer.com/10.1007/11691372\\_{ }33](http://link.springer.com/10.1007/11691372_{ }33). 28
- Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5673 LNCS, pages 69–85, 2009. ISBN 3642032362. doi:10.1007/978-3-642-03237-0\_7. URL [http://link.springer.com/10.1007/978-3-642-03237-0\\_{ }7](http://link.springer.com/10.1007/978-3-642-03237-0_{ }7). 28
- Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Jochen Hoenicke, Markus Lindenmann, Betim Musa, Christian Schilling, Stefan Wissert, and Andreas Podelski. Ultimate automizer with unsatisfiable cores (Competition contribution). In

- Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8413 LNCS, pages 418–420, 2014. ISBN 9783642548611. doi:10.1007/978-3-642-54862-8\_35. URL [http://link.springer.com/10.1007/978-3-642-54862-8\\_{ }35](http://link.springer.com/10.1007/978-3-642-54862-8_{ }35). 29
- Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. pages 168–176. 2004. ISBN 1045389020279. doi:10.1007/978-3-540-24730-2\_15. URL [http://link.springer.com/10.1007/978-3-540-24730-2\\_{ }15](http://link.springer.com/10.1007/978-3-540-24730-2_{ }15). 32
- Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, jul 2012. ISSN 00985589. doi:10.1109/TSE.2011.59. URL <http://arxiv.org/abs/0907.2072http://ieeexplore.ieee.org/document/5928354/>. 33
- Daniel Kroening and Michael Tautschnig. CBMC - C Bounded Model Checker (Competition contribution). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8413 LNCS, pages 389–391, 2014. ISBN 9783642548611. doi:10.1007/978-3-642-54862-8\_26. URL [http://svn.cprover.org/svn/cbmc/releases/cbmc-4.5-sv-comp-2014/.http://link.springer.com/10.1007/978-3-642-54862-8\\_{ }26](http://svn.cprover.org/svn/cbmc/releases/cbmc-4.5-sv-comp-2014/.http://link.springer.com/10.1007/978-3-642-54862-8_{ }26). 33
- Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An Industrial-strength C Model Checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, New York, New York, USA, oct 2018. ACM Press. ISBN 978-1-4244-0978-5. doi:10.1109/ISWCS.2007.4392451. URL <http://ieeexplore.ieee.org/document/4392451/http://dl.acm.org/citation.cfm?doid=3238147.3240481http://doi.acm.org/10.1145/3238147.3240481>. 34
- Peter Schrammel and Daniel Kroening. 2LS for program analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9636, pages 905–907, 2016. ISBN 9783662496732. doi:10.1007/978-3-662-49674-9\_56. URL [http://link.springer.com/10.1007/978-3-662-49674-9\\_{ }56](http://link.springer.com/10.1007/978-3-662-49674-9_{ }56). 34, 36
- Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977. ISSN 00900036. doi:10.1145/512950.512973. 35
- Cristina David, Daniel Kroening, and Matt Lewis. Second-Order Propositional Satisfiability. *CoRR*, abs/1409.4, sep 2014. URL <http://arxiv.org/abs/1409.4925>. 35
- Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In Sandrine Blazy and

- Thomas Jensen, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9291 of *Lecture Notes in Computer Science*, pages 145–161, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 9783662482872. doi:10.1007/978-3-662-48288-9\_9. URL [http://link.springer.com/10.1007/978-3-662-48288-9http://link.springer.com/10.1007/978-3-662-48288-9\\_{ }9](http://link.springer.com/10.1007/978-3-662-48288-9http://link.springer.com/10.1007/978-3-662-48288-9_{ }9). 35, 85
- Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8559 LNCS, pages 106–113, 2014. ISBN 9783319088662. doi:10.1007/978-3-319-08867-9\_7. URL [http://link.springer.com/10.1007/978-3-319-08867-9\\_{ }7](http://link.springer.com/10.1007/978-3-319-08867-9_{ }7). 36
- Mike Barnett, BY Chang, R DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Formal Methods for ...*, pages 364–387, 2006. ISSN 0-7695-1950-4. doi:10.1007/11804192\_17. URL <http://www.springerlink.com/index/d5618027k6v50257.pdf>. 36
- Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A Whole-Program Analyzer for Boogie. page 78, 2011. 36
- Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification*, pages 504–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73367-6. doi:10.1007/978-3-540-73368-3\_51. URL [http://link.springer.com/10.1007/978-3-540-73368-3\\_{ }51](http://link.springer.com/10.1007/978-3-540-73368-3_{ }51). 37
- Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, pages 29–38. IEEE, sep 2008. ISBN 9781424421886. doi:10.1109/ASE.2008.13. URL <http://ieeexplore.ieee.org/document/4639306/>. 37
- Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 189–197, 2010. ISBN 978-1-4577-0734-6. 37
- Dirk Beyer and M. Erkan Keremoglu. CPAChecker: A tool for configurable software verification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6806 LNCS, pages 184–190, 2011. ISBN 9783642221095. doi:10.1007/978-3-642-22110-1\_16. URL [http://link.springer.com/10.1007/978-3-642-22110-1\\_{ }16](http://link.springer.com/10.1007/978-3-642-22110-1_{ }16). 37
- Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *Lecture Notes in Computer Science (including*

- subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*), volume 8044 LNCS, pages 36–52, 2013. ISBN 9783642397981. doi:10.1007/978-3-642-39799-8\_2. URL [http://link.springer.com/10.1007/978-3-642-39799-8\\_{\\_}2](http://link.springer.com/10.1007/978-3-642-39799-8_{_}2). 39
- Matthias Heizmann, Yu Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants: (Competition contribution). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10806 LNCS, pages 447–451, 2018. ISBN 9783319899626. doi:10.1007/978-3-319-89963-3\_30. URL [http://link.springer.com/10.1007/978-3-319-89963-3\\_{\\_}30](http://link.springer.com/10.1007/978-3-319-89963-3_{_}30). 39
- Matthias Heizmann, Daniel Dietsch, Jan Leike, Betim Musa, and Andreas Podelski. ULTIMATE AUTOMIZER with array interpolation (Competition contribution). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9035, pages 455–457, 2015. ISBN 9783662466803. doi:10.1007/978-3-662-46681-0\_43. URL [http://link.springer.com/10.1007/978-3-662-46681-0\\_{\\_}43](http://link.springer.com/10.1007/978-3-662-46681-0_{_}43). 39
- Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-642-08474-4. doi:10.1007/978-3-662-03811-6. URL <http://link.springer.com/10.1007/978-3-662-03811-6>. 41
- Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. *Software Engineering and Formal Methods*, pages 233–247, 2012. doi:10.1007/978-3-642-33826-7\_16. URL [http://link.springer.com/10.1007/978-3-642-33826-7\\_{\\_}16](http://link.springer.com/10.1007/978-3-642-33826-7_{_}16)[http://link.springer.com/chapter/10.1007/978-3-642-33826-7\\_{\\_}16](http://link.springer.com/chapter/10.1007/978-3-642-33826-7_{_}16). 41
- Paul Anderson and Tim Teitelbaum. Software inspection using codesurfer. . . . *Inspection in Software Engineering (CAV 2001 . . . , 2001*. 41
- Mark Weiser. Program slicing. *Proceedings of the 5th international conference on Software engineering*, 1981. ISSN 0098-5589. doi:10.1109/TSE.2005.112. 42
- Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for C programs. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009*, pages 123–124. IEEE, 2009. ISBN 9780769537931. doi:10.1109/SCAM.2009.22. URL <http://ieeexplore.ieee.org/document/5279933/>. 42
- Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard Version 2.0. *8th International Workshop on Satisfiability Modulo Theories*, 2010. ISSN 09277757. doi:10.1016/S0927-7757(97)00232-X. 44

- Sylvia Wassertheil and Jacob Cohen. Statistical Power Analysis for the Behavioral Sciences. *Biometrics*, 26(3):588, sep 1970. ISSN 0006341X. doi:10.2307/2529115. URL <https://www.jstor.org/stable/2529115?origin=crossref><http://www.jstor.org/stable/2529115?origin=crossref>. 52
- José G. Pérez-Silva, Miguel Araujo-Voces, and Víctor Quesada. NVenn: Generalized, quasi-proportional Venn and Euler diagrams. In Jonathan Wren, editor, *Bioinformatics*, volume 34, pages 2322–2324, jul 2018. ISBN 13674811 (Electronic). doi:10.1093/bioinformatics/bty109. URL <https://academic.oup.com/bioinformatics/article/34/13/2322/4904268>. 55
- John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, page 335, New York, New York, USA, 2012. ACM Press. ISBN 9781450312059. doi:10.1145/2254064.2254104. URL <http://dl.acm.org/citation.cfm?doid=2254064.2254104>. 57
- Student. The Probable Error of a Mean. *Biometrika*, 6(1):1, mar 1908. ISSN 00063444. doi:10.1093/biomet/6.1.1. URL <https://academic.oup.com/biomet/article-lookup/doi/10.1093/biomet/6.1.1><https://www.jstor.org/stable/2331554?origin=crossref>. 58
- J. D. Lee and K. A. See. Trust in Automation: Designing for Appropriate Reliance. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 46(1): 50–80, jan 2004. ISSN 0018-7208. doi:10.1518/hfes.46.1.50\_30392. URL [http://hfs.sagepub.com/cgi/doi/10.1518/hfes.46.1.50\\_{\\_}30392](http://hfs.sagepub.com/cgi/doi/10.1518/hfes.46.1.50_{_}30392). 75
- Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, aug 2010. ISSN 15678326. doi:10.1016/j.jlap.2010.03.012. URL <http://linkinghub.elsevier.com/retrieve/pii/S1567832610000160>. 76
- Felix Neubauer, Karsten Scheibler, Bernd Becker, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, and Detlef Fehrer. Accurate dead code detection in embedded C code by arithmetic constraint solving. In *CEUR Workshop Proceedings*, volume 1804, pages 32–38, 2016. doi:10.1207/s15327809jls0403. 84, 86
- Marek Chalupa, Martina Vitovská, Martin Jonáš, Jiri Slaby, and Jan Strejček. *Symbiotic 4: Beyond reachability*, volume 10206 LNCS. 2017. ISBN 9783662545799. doi:10.1007/978-3-662-54580-5\_28. URL [http://link.springer.com/10.1007/978-3-662-54580-5\\_{\\_}28](http://link.springer.com/10.1007/978-3-662-54580-5_{_}28). 85
- Kumar Madhukar, Bjorn Wachter, Daniel Kroening, Matt Lewis, and Mandayam Sivas. Accelerating invariant generation. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design, FMCAD 2015*, pages 105–111. IEEE,

sep 2016. ISBN 9780983567851. doi:10.1109/FMCAD.2015.7542259. URL <http://ieeexplore.ieee.org/document/7542259/>. 86

Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. MUX: algorithm selection for software model checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 132–141, New York, New York, USA, 2014. ACM Press. ISBN 9781450328630. doi:10.1145/2597073.2597080. URL <http://dl.acm.org/citation.cfm?doid=2597073.2597080>. 86