Logical Methods in Automated Hardware and Software Verification

HABILITATION THESIS

submitted to the
Faculty of Informatics of TU Wien
in February 2016 by

Georg Weissenbacher

to Helmut Veith
mentor, role model, and friend

Abstract

Computer systems control the world – we have come to rely on the correctness of software and integrated circuits in safety-critical domains such as aviation and automotive engineering, as well as in communication, consumer electronics, and domestic appliances. Yet establishing the correctness of these systems is an ever more tedious and challenging task, which more often than not involves a significant manual effort.

Formal verification techniques such as model checking promise remedy in the form of fully automated tools that answer a range of questions about the system: does it behave as it is supposed to, or does it contain bugs buried deeply in the code? If the result is not as expected, exactly when and where did the system start to misbehave?

In practice, the scale of industrial software and hardware designs and the complexity of bugs severely limits the applicability of automated tools. Software components comprise thousands of lines of code; bugs (such as buffer overflows) may require hundreds of loop iterations to surface; and failing executions of integrated circuits can span millions of cycles.

The constituent publications of this habilitation thesis present novel techniques—centered around mathematical induction and Craig's interpolation theorem—that push the scalability of automated verification to real-world problems. We combine the most competitive hardware model checking algorithms (which are based on inductive generalization and interpolation) with novel abstraction-refinement techniques to verify industrial-size software. We show how to uncover deep software bugs by collapsing millions of execution steps into a single step by solving recurrences. Using interpolation, we isolate faults in lengthy executions in post-silicon validation. Our novel adaptations of induction and interpolation methods are key to the development of scalable tools for automated verification, bug detection, and fault localization.

Acknowledgments

The decade since I started my doctoral studies at ETH Zürich has been the most exciting of my life so far. I had the honor to work with numerous brilliant people without whom this habilitation thesis would not exist. First and foremost I have to thank the people who taught me how to do research – my academic supervisors and advisors Roderick Bloem, Daniel Kröning, Sharad Malik, and Helmut Veith (in alphabetical and chronological order). The publications that form this cumulative habilitation, however, owe their existence not only to my advisors but also to my numerous co-authors. In particular, I want to thank Johannes Birgmeier, Aaron Bradley, Vijay D'Silva, Matt Lewis, Mitra Purandare, Matthias Schlaipfer, Yakir Vizel, and Charlie Shucheng Zhu for their contributions. My work has been made possible by a number of funding agencies, in particular by the Vienna Science and Technology Fund (WWTF), the European Union (through FP7), the Austrian Science Fund (FWF), and Microsoft Research. Last but not least, I want to credit and express my gratitude to the people who selflessly supported my academic career: my parents, and my amazing wife (and copy editor) May.

Contents

Part I

Overview

1 | Introduction

Over the last few years, a staggering number of reports of bugs in software and hardware have been published by mainstream news outlets such as *The Guardian* or *The New York Times*. Prominent examples (some of which are listed in Figure 1.1) include a critical software bug causing a complete electrical shutdown of the Boeing 787 Dreamliner [Gib15]; the "Heartbleed" bug in the OpenSSL cryptographic library which puts users' personal information at risk [Kel14]; and the "Rowhammer" security vulnerability which allows attackers to take over a computer system by exploiting a hardware glitch in DRAM memory cells [Kir15].

These bugs and vulnerabilities remained undetected during product development despite huge efforts spent on verification and validation: more than 50 percent of time and cost of typical programming projects is dedicated to verification [MSB11], and a recent study [Fos12] suggests that the situation in hardware development is similar. Verification is increasingly becoming the bottle-neck in the development of electronic systems, since tasks such as testing and test-bench development, simulation, and debugging involve substantial manual effort.

Formal verification techniques such as model checking promise remedy in the form of fully automated tools. Model checking [CGP99] is an automated approach to exhaustively exploring and analyzing the behavior of systems. Typically applied to source code of software or hardware designs, model checking tools check for a range of bugs such as assertion violations, deadlocks, or crashes. Our surveys on software and hardware model checking ([DKW08] and [VWM15], respectively) provide an overview of the state-of-the-art in the field.

In practice, the scale of industrial software and hardware designs and the complexity of bugs severely limit the applicability of automated tools. For instance, the OpenSSL implementation comprises several hundred thousand lines of code only few of which are responsible for the Heartbleed bug; the shutdown of the 787 Dreamliner is caused by an arithmetic integer overflow which only surfaces after 248 days of continuous operation; the Rowhammer vulnerability is a hardware deficiency introduced during the manufacturing process and not reflected by a high-level hardware model. While these scenarios pose unique challenges to automated verification tools, the underlying problem is ultimately related to scalability and needs to be addressed by techniques that allow us to automatically analyze larger and more complex code and longer executions.

The publications collected in this habilitation thesis describe work undertaken at TU Wien between 2012 and 2015. The aim of this line of work is to increase the scalability of automated verification using novel techniques centered around mathematical induction and Craig's interpolation theorem.

“ Toyota Motor is recalling all of the 1.9 million newest-generation Prius vehicles it has sold worldwide because of a programming error ... ”

The New York Times, Feb. 13, 2014 [TT14]

“ The US air safety authority has issued a warning and maintenance order over a software bug that causes a complete electric shutdown of Boeing's 787 ... ”

The Guardian, May 1, 2015 [Gib15]

“ A major online security vulnerability dubbed “Heartbleed” could put your personal information at risk, including passwords, credit card information and e-mails. ”

CNN, Apr. 9, 2014 [Kel14]

“ An unprecedented systems failure was responsible for the air traffic control chaos that affected airports across London and southeast England on Friday ... “In this instance a transition between the two states caused a failure in the system which has not been seen before,” ... ”

The Guardian, Dec. 13, 2014 [Joh14]

“ DRAM is vulnerable to electrical interference because the cells are packed so closely together ...

Repeatedly accessing a row of memory cells can cause adjacent ones to change their binary values using a technique that's been described as rowhammering. ”

InfoWorld, July 30, 2015 [Kir15]

Figure 1.1: Software bugs and hardware faults in recent news

The habilitation thesis is structured as follows:

Part I, comprising the first three chapters of this thesis, describes the relevant background and establishes the overarching theme of the constituent publications in Part II.

Chapter 2 provides an overview of modern verification and model checking techniques, including symbolic model checking, abstraction and refinement, and acceleration.

Chapter 3 summarizes the contributions of the papers that constitute Part II of the thesis and relates their content. The techniques covered by these papers fall into four categories:

1. State-of-the-art model checking algorithms and foundations of algorithmic Craig interpolation, which form the basis of the work presented in the subsequent chapters;
2. Abstraction and refinement techniques for software model checking, which rely on induction and interpolation to increase the scalability of software model checkers;

3. Acceleration and under-approximation techniques (which are based on induction) for the rapid detection of software bugs; and
4. Interpolation-based hardware fault localization techniques, enabling the efficient localization of faults in lengthy executions of integrated circuits.

Part II comprises the constituent publications of the habilitation thesis.

Chapter 4 provides a survey of state-of-the-art hardware model checking techniques and their underlying satisfiability checking algorithms [VWM15]. The primary focus of the survey is on verification techniques based on Craig interpolation (cf. Chapter 5) and induction (such as the IC3 model checking paradigm, which we apply to software in Chapter 6).

Chapter 5 describes novel algorithms to construct Craig interpolants [SW16] from propositional resolution proofs and first-order logic proofs (generated by solvers surveyed in Chapter 4), generalizing a range of existing interpolation techniques.

Chapter 6 combines IC3—one of the the most competitive hardware model checking algorithms (discussed in Chapter 4)—with interpolation-based abstraction techniques to verify industrial-size software [BBW14].

Chapter 7 explains how acceleration techniques based on mathematical induction can be applied to rapidly detect bugs in software in the presence of loops with many iterations and datatypes such as bit-vectors and arrays [KLW15b].

Chapter 8 discusses how the acceleration techniques from Chapter 7 can be deployed to quickly prove the absence of "deep" bugs that require many loop iterations [KLW15a].

Chapter 9 describes how Craig interpolation can be deployed to enable an iterative analysis of lengthy executions of integrated circuits to locate hardware faults introduced during the manufacturing process [ZWM14].

The work presented in the peer-reviewed articles in Chapters 4 to 9 has been carried out in close collaboration with a range of excellent students and researchers, to whom I owe a great debt. As described below, my contributions to these publications were significant.

- The survey in Chapter 4 is joint work with my former post-doctoral advisor Sharad Malik (at Princeton University), who provided the vision and outline for the paper, and Yakir Vizel, to whom the part on interpolation-based algorithms owes its existence.
- The journal paper in Chapter 5 is an extended version of my single-authored work [Wei12], which was enriched with additional results and experiments on interpolation for clausal refutations by my doctoral student Matthias Schlaipfer.
- The paper on our adaptation of the IC3 model checking algorithm (Chapter 6) for software resulted from a visit of Aaron Bradley (the inventor of IC3) and is based on the thesis of my then master's student Johannes Birgmeier (now a doctoral student at Stanford).

- Chapters 7 and 8 are joint work with my doctoral supervisor Daniel Kröning and his student Matt Lewis, and a direct continuation of a line of work I started during my doctorate [KW06, KW10, Wei10]. The implementation is based on my software verification tool WOLVERINE [KW11].

- Chapter 9 is joint work with Sharad Malik and his doctoral student Charlie Shucheng Zhu, whom I co-supervised at Princeton. The implementation is based on a SAT-based tool for computing backbones that I developed [ZWSM11].

2 | Model Checking 101

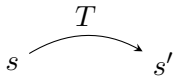


Figure 2.1: Transition relation T

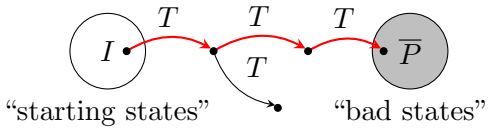
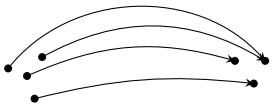
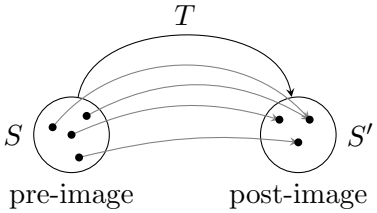


Figure 2.2: Counterexample to property P



(a) Single transitions and explicit states



(b) Binary relation T over sets of states

Figure 2.3: Explicit exploration vs. image computation

Model checking [CE81, QS82, CGP99] is an automated verification technique that exhaustively checks a system for bugs (or violations of a given specification). Unlike testing, model checking is not based on executing the system under test, but rather on a systematic analysis of a *model* of the system. In its most general form, the model of a system is given in terms of a transition relation T relating states s to their successor states s' (Figure 2.1). At this level of abstraction, we are oblivious of the nature of the modeled system: states could be either program states mapping memory locations and program counters to values (e.g., $\langle pc \mapsto 2, x \mapsto 1 \rangle$), the states of the latches of an integrated circuit, or a more abstract system description such as a state machine or a business model described in the Unified Modeling Language (UML).

The goal of model checking is to determine whether a given system satisfies a property in question. In the simplest case, the property is given as a set of safe states P that the system must not leave (or, alternatively, a set of bad states $\overline{P}$ that the system should never reach). Consequently, model checking algorithms are search algorithms that systematically explore states reachable from an initial set of states I. If the search algorithm encounters states for which T yields no successors or states that have been previously visited, it backtracks and explores undiscovered regions of the state space. A sequence of transitions violating the property is a *counterexample* to safety (indicated in red in Figure 2.2).

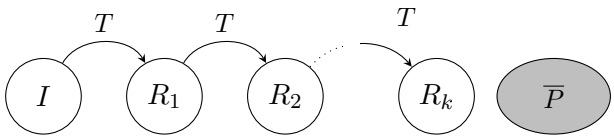


Figure 2.4: A k-step exploration of reachable sets of states

In practice, the number of reachable states (the state space) is often prohibitively large, posing an insurmountable obstacle to naïve search algorithms (a central problem in automated verification known as state space explosion). Approaches based on the explicit (one-by-one) enumeration of states—therefore named explicit-state model checking—are particularly prone to the combinatorial explosion of the number of states. Rather than considering states and their successors individually, one can consider sets of states and their post-image under the transition relation T:

$$S' \quad = \quad T(S) \stackrel{\text{def}}{=} \{s' \,|\, T(s, s') \wedge s \in S\} \quad (2.1)$$

The post-image of S under T is illustrated in Figure 2.3.[1]

In this setting, state space search amounts to a sequence of computations of post-images of T starting from the initial set of states I. Figure 2.4 illustrates the exploration of the state space up to a depth of k execution steps: each set R_i represents the states reachable by exactly i steps or the transition relation T.

[1]Note that S and S' are not necessarily disjoint, as the domain and co-domain of T coincide.

2.1 Symbolic Encodings

The success of a set-based exploration hinges on an efficient representation of sets of states (as a simple container of single states offers no significant advantage over explicit-state model checking). Symbolic model checking [BCM+90, McM93], an approach that marked a major breakthrough in the scalability of hardware model checking algorithms, uses Binary Decision Diagrams (BDDs) [Bry86] (a graph-based data structure to represent propositional formulas) to encode sets of states. Similarly, first-order predicates provide a compact symbolic representation of sets of program states [Kin70] frequently used in software model checking (e.g., [BCLR04, Jha04, McM06a]). A first-order predicate P over the variables V of a program encodes all states in which P evaluates to true. For instance,

$$(x > 0) \quad \text{represents} \quad \{s \,|\, s(x) > 0\}\,,$$

i.e., the set of program states s in which x is larger than zero and all other variables have arbitrary values. Thus, a single predicate can even encode an infinite set of program states.

Transition relations T are encoded as relations over the variables V and a set of primed variables $V' \stackrel{\text{def}}{=} \{v' \,|\, v \in V\}$ representing successor states. For instance, for a simple program with a single variable $V \stackrel{\text{def}}{=} \{x\}$, the statement `x++` is encoded as follows:

$$\underbrace{(x' = x + 1)}_{\texttt{x++}} \text{ represents } \{\langle s, s' \rangle \,|\, s'(x) = s(x) + 1\}$$

In case V is not a singleton, the constraint $\bigwedge_{y \in (V \setminus x)} (y' = y)$ over all remaining variables

```
1:  if (x>0)
2:    x = x - 1;
3:  else
4:    x = x + 1;
5:  assert (x≥0);
```

Figure 2.5: A simple program fragment

is added to ensure that the values of the variables unaffected by the statement `x++` remain unchanged.

To represent control-flow, a dedicated variable $pc \in V$ is introduced to represent the program counter. Figure 2.6 shows the symbolic encoding of the simple program fragment in Figure 2.5, which can be readily obtained from the source code. Each implication in Figure 2.6 encodes a transition from a program location to one of its successors in the control flow graph. For conditional statements, the premises ($x > 0$ or $x \leq 0$ in Figure 2.6a) determine which branch is taken. Assignment statements are executed unconditionally and update the state of the program variables (see Figure 2.6b). Initial states and safety properties are encoded as simple predicates over V (as in Figure 2.6c).

Digital circuits, such as the simple sequential circuit in Figure 2.7 (a Mealy machine, in which the output signal z is determined by the current state Q and the current input signal y) are encoded in a similar manner. The propositional variables V represent latches as well as input and output signals of the circuit:

$$\big(Q' \Leftrightarrow (x \wedge Q)\big) \wedge (z \Leftrightarrow (y \vee Q))$$

Note that the primed counterparts x', y', and z' of the input and output signals are unconstrained, since the inputs of the next cycle are determined externally rather than by the circuit.

$$\begin{aligned} (pc = 1) \wedge \quad (x > 0) &\Rightarrow \quad (pc' = 2) \wedge (x' = x) \\ (pc = 1) \wedge \neg(x > 0) &\Rightarrow \quad (pc' = 4) \wedge (x' = x) \end{aligned}$$

(a) Conditional statement

$$\begin{aligned} (pc = 2) &\Rightarrow \quad (pc' = 5) \wedge (x' = x - 1) \\ (pc = 4) &\Rightarrow \quad (pc' = 5) \wedge (x' = x + 1) \end{aligned}$$

(b) Assignment statements

$$\begin{aligned} P(V) \quad &\stackrel{\text{def}}{=} \quad (pc = 5) \Rightarrow (x \geq 0) \\ I(V) \quad &\stackrel{\text{def}}{=} \quad (pc = 1) \end{aligned}$$

(c) Initial states and property

Figure 2.6: Symbolic encoding of Figure 2.5

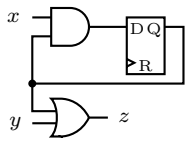


Figure 2.7: A simple sequential circuit

The same mechanism is applied in software to model inputs.

A transition $T(V, V')$ represents the execution of a single program statement or a single cycle of the circuit, respectively. An execution of k instructions or cycles can be encoded by means of k copies of the transition relation T over $k+1$ versions of the variables V:

$$I(V_0) \wedge \left(\bigwedge_{i=1}^{k} T(V_{i-1}, V_i) \right) \wedge \neg P(V_k) \qquad (2.2)$$

The resulting Formula (2.2) represents all executions of length k which violate the property P after k steps (as illustrated in Figure 2.4). Contemporary decision procedures for propositional or first-order logic (discussed in our survey [VWM15] in Chapter 4 or [KS08, BHvMW09],

for instance) can determine whether Formula (2.2) is satisfiable, and thus whether there exists an execution violating the property after k steps. This approach, known as Bounded model checking (BMC) [BCCZ99], was first implemented for hardware based on efficient satisfiability checkers for propositional logic (cf. [VWM15]/Chapter 4). Implementations for software are described in [CKL04] and our survey [DKW08].

BMC (in the form presented in Formula (2.2)) is inherently limited to detecting property violations and incapable of ultimately proving a model safe. The reason for this incompleteness of BMC is that the approach lacks a mechanism to determine whether all reachable states have been explored. While there are means to determine whether a model has been explored in sufficient depth [CKOS04] or whether subsequent unwinding steps of T are still feasible (see [CKL04] and our publication [KLW15a] in Chapter 8), these techniques are typically either inefficient (in the sense that they vastly overestimate the bound) or incomplete. Complete model checking algorithms, capable of establishing the safety of a model are predominantly based on computing all reachable states of a model (or an approximation thereof).

2.2 Invariants and Acceleration

The set R_k of states reachable from the initial states I in k steps of T is defined inductively as follows:

$$\begin{array}{lcl} R_0 & \stackrel{\text{def}}{=} & I \\ R_{i+1} & \stackrel{\text{def}}{=} & T(R_i) \end{array} \tag{2.3}$$

A solution to Equation (2.3) can be determined by iteratively computing the image of R_i under T (as defined in Equation (2.1)). In the context of a symbolic encoding of T as introduced in Section 2.1, image computation corresponds to existential quantification:

$$S'(V') \quad \stackrel{\text{def}}{=} \quad \exists V \,.\, S(V) \wedge T(V, V') \tag{2.4}$$

Using Equation (2.4), the set $R_{\leq k}$ of states reachable from $R_{\leq 0} \stackrel{\text{def}}{=} I(V_0)$ in k or less steps of T can be obtained by an iterative and cumulative image computation:

$$R_{\leq (i+1)}(V_{i+1}) \quad \stackrel{\text{def}}{=} \quad \left(\begin{array}{c} R_{\leq i}(V_i)[V_{i+1}/V_i] \\ \vee \\ \exists V_i \,.\, R_{\leq i}(V_i) \wedge T(V_i, V_{i+1}) \end{array} \right), \tag{2.5}$$

where $R_{\leq i}(V_i)[V_{i+1}/V_i]$ denotes the formula obtained by replacing all free occurrences of the variables V_i in $R_{\leq i}(V_i)$ with their respective counterparts in V_{i+1}.

The search can stop if either $R_{\leq i}(V_i)$ does not imply $P(V_i)$ for some $i \geq 0$ (i.e., a bad state is reachable), or

$$\forall V_{i+1} \,.\, R_{\leq (i+1)}(V_{i+1}) \Rightarrow R_{\leq i}(V_i)[V_{i+1}/V_i] \tag{2.6}$$

holds, meaning that no new states can be reached from $R_{\leq i}(V_i)$ via T. In other words, if Equation (2.6) holds, then $R_{\leq i}$ (representing the exact set of states reachable from I) is the least fixpoint of T.

Determining the least fixpoint can be computationally expensive: Equation (2.6) contains alternating quantifiers, since $R_{\leq (i+1)}$ is existentially quantified in Equation (2.5), posing a difficult problem for contemporary decision procedures. To determine whether a system is safe, however, an over-approximation Inv of the least fixpoint suffices, as long as it represents a set of safe states that includes the initial set of states I and from which one "cannot escape" via T (see

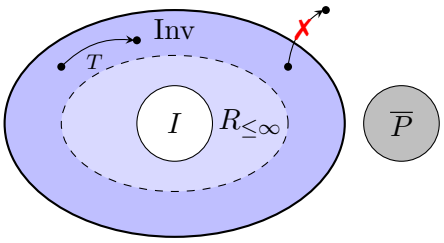


Figure 2.8: Inductive invariant

Figure 2.8). For every state in Inv, its successor (determined by T) must also lie in Inv. Consequently, if Inv contains I, then Inv must also encompass the set of all states $R_{\leq\infty}$ reachable from I (i.e., the least fixpoint).

These conditions can be formalized as follows:

$$\begin{array}{rcll} I(V) & \Rightarrow & \mathrm{Inv}(V) & \\ \exists V \,.\, \mathrm{Inv}(V) \wedge T(V,V') & \Rightarrow & \mathrm{Inv}(V') & (2.7) \\ \mathrm{Inv}(V) & \Rightarrow & P(V) & \end{array}$$

where $\mathrm{Inv}(V')$ is short for $\mathrm{Inv}(V)[V'/V]$. Inv is an *inductive* invariant (and a post-fixpoint of T, which guarantees that $R_{\leq i}(V) \Rightarrow \mathrm{Inv}(V)$ for all $i \geq 0$) that is sufficiently accurate to guarantee that property P is not violated.

Consequently, many contemporary model checking algorithms are based on the search for an inductive invariant and deploy a number of heuristics to obtain appropriate approximations of the reachable states. State-of-the-art techniques (predominantly based on interpolation and induction) are discussed in Section 3.1 as well as Chapters 4 and 6 ([VWM15] and [BBW14], respectively).

An alternative approach to computing reachable states is by means of acceleration, a technique used to derive a new transition relation T^* which subsumes arbitrarily many steps of T (the reflexive and transitive closure of T). Intuitively, acceleration allows us to take a short-cut by collapsing arbitrarily many steps of T into one step, T^*. Formally (based on the notation in Equation 2.1),

$$\begin{array}{rcl} T^*(S) & \stackrel{\text{def}}{=} & \bigcup_{i\in\mathbb{N}} T^i(S), \text{ where} \\ T^{i+1}(S) & \stackrel{\text{def}}{=} & T(T^i(S)) \text{ and} \\ T^0(S) & \stackrel{\text{def}}{=} & S \end{array} \qquad (2.8)$$

The goal of acceleration [BW94, Boi99, FL02] is to compute T^* symbolically, e.g., by deriving it from a transition relation T given in linear arithmetic. An accelerated version of the above mentioned transition $T \stackrel{\text{def}}{=} (x' = x + 1)$, for instance, would be $T^* \stackrel{\text{def}}{=} (\exists k \,.\, x' = x + k)$, where k is a fresh variable encoding the number of steps of T. Note that T^* simulates arbitrarily many steps of T, reducing the question of whether P can be violated by T to whether P can be violated in a single step of T^*:

$$I(V_0) \wedge T^*(V_0, V_1) \wedge \neg P(V_1)$$

In general, however, whether T^* can not always be expressed as a logical relation, even if T is defined as a relation in a decidable fragment of arithmetic (such as Presburger arithmetic). Accordingly, acceleration is typically applied to transition relations formulated in simple logical fragments (e.g., octagonal relations of the form $\pm x \pm y \leq c$).

Moreover, software programs and circuit designs usually use bit-vectors (of bounded width) to represent the integers, introducing non-linear effects such as arithmetic overflows. In such a setting, the above mentioned acceleration techniques based on linear arithmetic are unsound, as the accelerated transitions do not accurately reflect the behavior of the program. The papers [KLW15b] and [KLW15a] in Chapters 7 and 8 present a novel approach that enables acceleration in the presence of realistic datatypes such as bit-vectors and arrays.

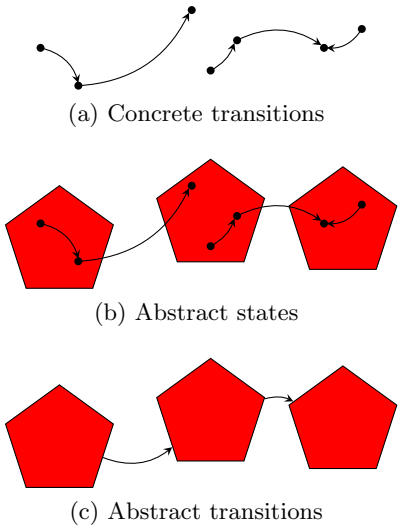
2.3 Abstraction

Abstraction [CGL92, Kur94] is arguably the most powerful and influential paradigm in model checking. It relies on the fact that typically not all details of a model need to be taken into account to prove its safety. Abstraction heavily relies on heuristics to identify irrelevant details of the system that are not required to verify that the property P in question holds. Proof-based abstraction [MA03], for instance, starts with an unsatisfiable instance of Formula (2.2) which shows that P cannot be violated in k steps, and uses the ability of contemporary satisfiability solvers to determine an unsatisfiable *core* of Formula (2.2) that contains the details (variables, latches, ...) required to obtain this conditional safety result. In predicate abstraction [GS97], a set of first-order predicates (encoding relevant details) induces a partitioning of the states of the model. For instance, two predicates $(x \leq y)$ and $(y \neq 0)$ result in four equivalence classes,

$$(x \leq y) \wedge (y \neq 0), (x > y) \wedge (y \neq 0),$$
$$(x \leq y) \wedge (y = 0), \text{ and } (x > y) \wedge (y = 0)\,,$$

each of which represents the set of states in which the respective Boolean combination of predicates evaluates to true. The states $\langle x \mapsto 0, y \mapsto 1\rangle$ and $\langle x \mapsto 1, y \mapsto 2\rangle$ are in $(x \leq y) \wedge (y \neq 0)$, for instance, while $\langle x \mapsto 1, y \mapsto 0\rangle$ and $\langle x \mapsto 2, y \mapsto 0\rangle$ are in $(x > y) \wedge (y = 0)$. Each equivalence class corresponds to an *abstract* state $\widehat{s}$ subsuming a set of *concrete* states (denoted by $\gamma(\widehat{s})$, where γ is a so-called concretization function). Each concrete state s has a corresponding abstract state $\widehat{s} = \alpha(s)$, where α is the abstraction function induced by the equivalence classes.

Figure 2.9b illustrates 3 abstract states that



(a) Concrete transitions

(b) Abstract states

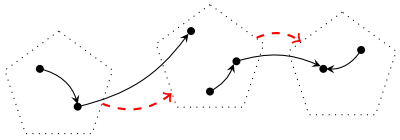
(c) Abstract transitions

Figure 2.9: Existential abstraction

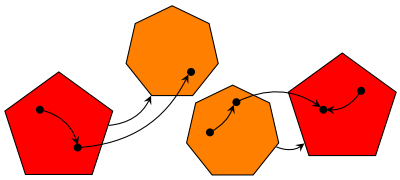
subsume the 7 concrete states in Figure 2.9a. Figure 2.9c shows the corresponding abstract transitions. The abstract transition relation $\widehat{T}$ enables a transition between two abstract states $\widehat{s}$ and $\widehat{s}'$ whenever there exist concrete states $s \in \gamma(\widehat{s})$ and $s' \in \gamma(\widehat{s}')$ such that $\langle s, s'\rangle \in T$. The relation between abstract and concrete transitions is illustrated by the following diagram:

$$\begin{array}{ccc} \widehat{s} & \xrightarrow{\widehat{T}} & \widehat{s}' \\ {\scriptstyle\alpha}\uparrow & & \downarrow{\scriptstyle\gamma} \\ s & \xrightarrow{T} & s' \end{array} \tag{2.9}$$

This approach, known as existential abstraction [CGL92], preserves all executions of the original model, but potentially adds more: in Figure 2.9c, for instance, the rightmost state is reachable from the leftmost state, which is not the case in Figure 2.9a. Abstraction potentially introduces *spurious* executions not present in the original model, and consequently, counterexamples may be spurious, too. Whether an



(a) Concretizing abstract executions



(b) Refined abstraction

Figure 2.10: Counterexample-guided refinement

abstract counterexample (represented by a sequence of abstract states $\widehat{s}_1, \ldots, \widehat{s}_k$) is spurious can be determined by checking whether there exists a corresponding concrete counterexample $s_1, \ldots, s_k$ such that $s_i \in \gamma(\widehat{s}_i)$ for $1 \leq i \leq k$ with $s_1 \in I$ and $s_k \in \overline{P}$, and $\langle s_i, s_{i+1} \rangle \in T$ for $1 \leq i < k$. Figure 2.10a illustrates this feasibility check for the abstraction in Figure 2.9.

Spurious counterexamples are undesirable since they result in false warnings. To eliminate a spurious counterexample, the precision of the abstraction needs to be improved (a process called refinement). The objective of refinement is to remove the spurious transition that connects two states in the spurious counterexample that are not connected by the original transition relation. The spurious counterexample can be used as trigger and guidance for refinement. In the example in Figure 2.10a, the concretization of the spurious counterexample provides sufficient information to determine which abstract state has to be refined (i.e., partitioned further). Figure 2.10b shows a refined model in which one of the states from the initial abstraction has been replaced with two (more precise) abstract states.

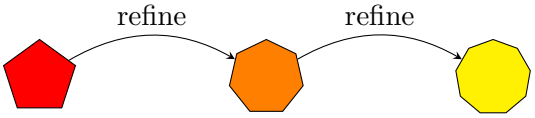


Figure 2.11: Repeated refinement steps

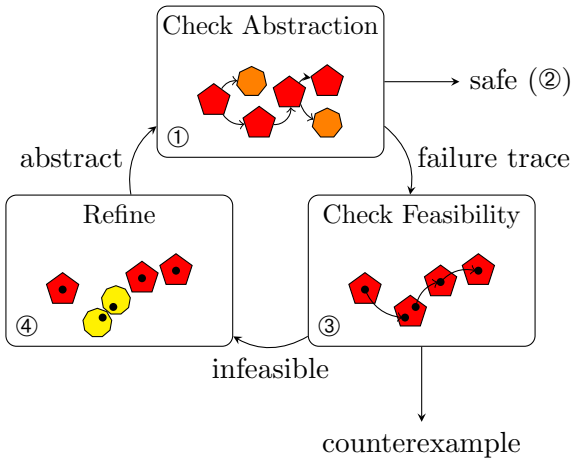


Figure 2.12: Counterexample-Guided Abstraction Refinement (CEGAR)

Counterexample-guided refinement (as illustrated in Figure 2.10) is based on an analysis of a spurious counterexample in the context of the original model. A commonly used heuristic to derive predicates required for refining a model is interpolation (introduced in [HJMM04] and described in Chapter 4/[VWM15] and our survey [DKW08]). Our publication [BBW14] in Chapter 6 describes a novel refinement technique which is based on single spurious transitions rather than entire counterexamples.

Repeated refinement steps lead to increasingly precise abstractions, as indicated in Figure 2.11. The resulting iterative abstraction-refinement scheme, known as Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ+00, CGJ+03], is illustrated in Figure 2.12:

1. CEGAR starts with a coarse initial abstraction, and the corresponding abstract model is analyzed by a model checker (①).

2. If the abstract model is safe, the original model must be safe, since abstraction only adds behaviors and therefore never removes concrete counterexamples (②).

3. If an abstract counterexample is found, its feasibility is checked (③). Counterexamples that have a feasible counterpart in the original model are reported; spurious counterexamples trigger a refinement step (④).

4. The process starts over with a refined model.

The CEGAR loop is not guaranteed to terminate, since the process is contingent on refinement heuristics. In predicate abstraction, for instance, a heuristic that fails to identify the relevant details of a model may yield a diverging sequence of predicates [KW06]. Yet, CEGAR has been successfully deployed in numerous verification tools (e.g., [BCLR04, HJMS02, McM06b]) and increases the scalability of model checking significantly.

The following chapter outlines the contributions of the papers presented in Part II of this habilitation. The techniques developed build heavily on the foundations discussed in the current section, and aim at improving the scalability of automated verification even further.

3 | Outline of Contributions

The following sections outline the main contributions of the papers presented in Chapters 4 to 9. These chapters describe state-of-the art model checking algorithms and their theoretical foundations (Section 3.1) as well as applications ranging from software verification to automated fault localization. The overarching theme is the use of Craig interpolation and induction to increase the scalability of automated verification.

3.1 Foundations

Over the last decade, automated decision procedures such as satisfiability (SAT) solvers [BHvMW09] and satisfiability-modulo-theory (SMT) solvers [KS08] have emerged as the cornerstone of modern verification tools. Decision procedures are algorithms that determine whether a given formula has a variable assignment that makes it true, a crucial step of all symbolic techniques presented in [VWM15] in Chapter 2. For example, any assignment *satisfying* a BMC instance—i.e., an assignment in which Formula (2.2) evaluates to true—corresponds to a counterexample of length k. Consequently, the scalability of BMC hinges on the efficiency of contemporary SAT and SMT solvers.

3.1.1 Interpolation-based Verification

The extensive role played by SAT solvers in hardware model checking is explained in our survey [VWM15] in Chapter 4. We give a comprehensive overview of modern satisfiability checking techniques, covering heuristics that are vital to scalability, as well as the generation of proofs and unsatisfiable cores, and Craig interpolation algorithms. While the latter techniques can be categorized as extensions of SAT solvers (described in further detail in our tutorial [MW12]), they form the basis of the model checking algorithms described in the second half of Chapter 4 (Section III of [VWM15]).

Contemporary model checking algorithms, covered in Section III of Chapter 4, rely heavily on approximation techniques that are based on (extensions of) decision procedures. At the core of many of these model checking algorithms lies Craig's interpolation theorem, which enables the computation of approximate post-images of transition relations.

Theorem 1 (Craig's Interpolation Theorem)**.** *Given two first-order formulas $A(V, V')$ and $B(V', V'')$ over the free variables $V \cup V' \cup V''$, the following holds:*

$$\textit{If } (A(V, V') \wedge B(V', V'')) \textit{ is unsatisfiable}$$
$$\textit{then}$$
$$\exists C(V')$$
$$\textit{such that}$$
$$A(V, V') \Rightarrow C(V')$$
$$B(V', V'') \Rightarrow \neg C(V')$$

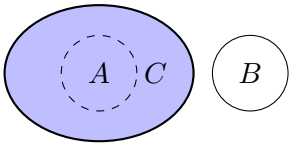


Figure 3.1: Formula C "generalizes" A

More verbosely, if the conjunction of A and B is unsatisfiable, then there exists an intermediate first-order formula C which is implied by A and inconsistent with B. Moreover, C refers only to the variables shared by A and B, meaning that it is in some sense "simpler" than A. The formula C can be understood as an over-approximation of A which comprises all satisfying assignments of A but none of B, as illustrated in Figure 3.1.

Theorem 1 can be readily applied to compute an approximate post-image of a transition relation T (cf. Equation 2.1). Assuming that property P is not violated in the first step of the transition relation T, the BMC instance

$$I(V) \wedge T(V, V') \wedge \neg P(V') \tag{3.1}$$

is unsatisfiable. A Craig interpolant $\mathrm{Itp}(V')$ for the partitions $I(V) \wedge T(V, V')$ and $\neg P(V')$ represents a safe over-approximation of the states R reachable from I in one step, as sketched in Figure 3.2. The formula Itp refers only to variables V', which represent the second time frame of the execution. Itp can be "shifted back in time" by variable renaming ($\mathrm{Itp}(V) \stackrel{\text{def}}{=} \mathrm{Itp}(V')[V/V']$).

Note that $I(V)$ is an inductive invariant (as specified in Equation 2.7) if $\mathrm{Itp}(V) \Rightarrow I(V)$. If this is not the case, we continue by using interpolation to compute an approximate post-image of $(I(V) \vee \mathrm{Itp}(V))$ under T – where $(I(V) \vee \mathrm{Itp}(V))$ represents a safe approximation of all states reachable in at most one step. Again, the system is safe if the resulting interpolant implies $(I(V) \vee \mathrm{Itp}(V))$.

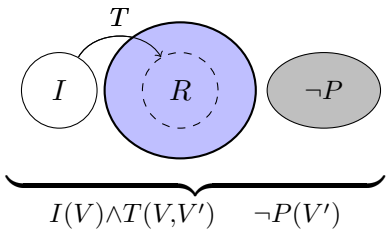


Figure 3.2: Interpolation-based image approximation

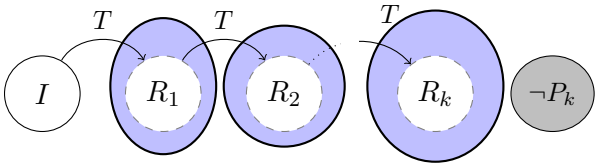


Figure 3.3: Interpolation sequences

As with the abstraction techniques presented in Section 2.3, over-approximation can result in spurious counterexamples. The precision of interpolation-based image approximation can be increased by generalizing the technique to k steps of T: a sequence of interpolants $\mathrm{Itp}_1, \ldots, \mathrm{Itp}_k$ over-approximating the reachable states $R_1, \ldots, R_k$ (as in Figure 3.3) can be obtained iteratively by splitting Formula (2.2)). Intuitively, the resulting interpolants are more precise, since a BMC instance of length k provides more information about the system than the single transition T in Figure 3.2. The system is safe if $\bigvee_{i=1}^{k} \mathrm{Itp}_i(V)$ forms an inductive invariant, and spurious counterexamples can be eliminated by increasing k.

Further variations and optimizations of interpolation-based model checking algorithms (as well as the induction-based IC3 model checking algorithm [Bra11], which we extend to soft-

ware in [BBW14] in Chapter 6) are discussed in Chapter 4. All these algorithms rely on the capability of solvers to effectively compute concise interpolants, a matter addressed in our journal paper [SW16] in Chapter 5 and the next subsection.

3.1.2 Interpolation Algorithms

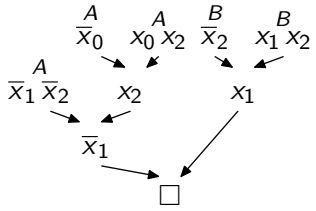
Most interpolating decision procedures derive interpolants from refutation proofs generated while solving the instance. Contemporary SAT solvers are capable of generating resolution proofs for unsatisfiable instances. Figure 3.4a, for instance, shows a resolution proof for the unsatisfiability of

$$\overbrace{(\neg x_1 \vee \neg x_2) \wedge (\neg x_0) \wedge (x_0 \vee x_2)}^{A} \wedge \underbrace{(\neg x_2) \wedge (x_1 \vee x_2)}_{B} . \quad (3.2)$$

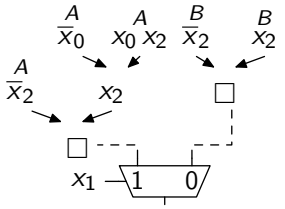
(Negations $\neg x$ in Figure 3.4a are abbreviated as $\overline{x}$ and the disjunctions in clauses are omitted for brevity.)

Interpolants can be understood as circuits following the structure of the resolution proof: an interpolant $\mathrm{Itp}(V')$ emits true for all inputs V' that make $A(V, V')$ true, and to false for all inputs V' that make $B(V', V'')$ true. Pudlák describes interpolants as a cascade of multiplexers (a gate selecting one of two signals depending on an input switch) whose input switches are the variables V' [Pud97], and provides a procedure to obtain interpolants by replacing resolution steps with logic gates, accordingly (as sketched in Figure 3.4b). We defer the details of the construction to [SW16] (Chapter 5).

Chapter 5 presents a range of novel interpolation techniques which generalize existing interpolation algorithms in the following sense:



(a) A resolution proof



(b) Interpolants as circuits

Figure 3.4: Interpolants from proofs

1. Our novel interpolation system is able to produce a wider range of interpolants and provides a means to fine-tune their logical strength and structure.

2. We are able to deal with a wider range of proof systems, including hyper-resolution proofs and clausal proofs that are generated by contemporary SAT solvers.

3. We generalize the results for propositional resolution systems to first-order proofs that satisfy certain locality properties (regarding the variables and symbols occurring in the proof).

Interpolating decision procedures form the basis of the verification techniques and applications presented in the subsequent chapters, including the generalization of the IC3 model checking algorithm to software, which we discuss in the following section.

3.2 Software Model Checking

To enable the verification of industrial-size code bases such as the OpenSSL library (known for its "Heartbleed" bug as discussed in Chapter 1) the scalability of contemporary model checking algorithms needs to be increased significantly. Our paper [BBW14] in Chapter 6 presents progress towards this goal by combining one of the most successful hardware model checking algorithms with predicate abstraction.

3.2.1 Model Checking with IC3

IC3 [Bra11] (short for "Incremental Construction of Inductive Clauses for Indubitable Correctness") is one of the leading hardware model checking algorithms (see Chapter 4). Similarly to interpolation-based verification techniques, IC3 is based on an over-approximation of reachable states. The algorithm maintains a sequence $F_0, \ldots, F_k$ of over-approximations of the sets $R_{\leq i}$, i.e., the states reachable in up to i steps, for $0 \leq i \leq k$.

Unlike the model checking algorithms described in Section 3.1.1, which are at the mercy of the underlying interpolating decision procedure when it comes to the accuracy of refinement, IC3 takes full control over the refinement of approximations. The algorithm steers refinement towards finding an invariant: intuitively, the refined approximation is "closer" to being an invariant. Recall from Section 2.2 that

(a) an invariant must not contain any state which has a "bad" successor in $\overline{P}$ (see Figure 2.8), and

(b) all states contained in an inductive invariant Inv must have their successors in Inv.

In IC3, refinement is driven by *counterexamples to induction* (CTI), states which are contained in the current over-approximation and from which $\overline{P}$ is reachable (i.e., which violate condition (a) above). For the system to be safe, each such CTI must be unreachable from the initial set of states I.

Assume that F_k (the approximation of $R_{\leq k}$, the set of states reachable in up to k steps) contains a CTI s, which is a predecessor of a state in $\overline{P}$ (as in Figure 3.5a). IC3 guarantees that s is not contained in F_{k-1}, since $\overline{P}$ would have otherwise been reached in a previous step.

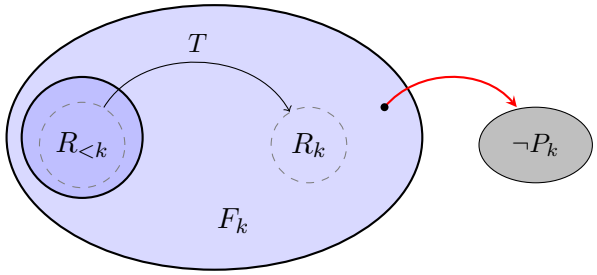
Using a symbolic encoding $s(V)$ of s, IC3 checks whether the CTI s is unreachable from F_{k-1} (the states reachable in up to $k-1$ steps):

$$\neg s(V) \wedge F_{k-1}(V) \wedge T(V, V') \stackrel{?}{\Rightarrow} \neg s(V')\,, \quad (3.3)$$

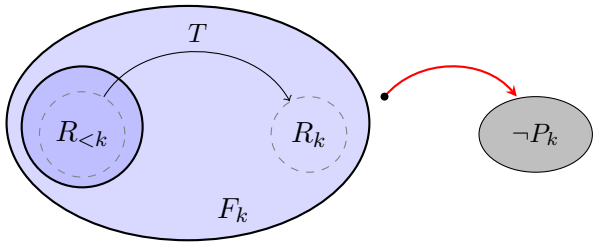
If Query (3.3) succeeds (i.e., the formula holds), then s (including similar states determined by generalization heuristics [HBS13]) is removed from F_k (Figure 3.5b). Moreover, the formula $\neg s(V)$ (and its generalization) is inductive *relative* to F_{k-1}: the image of $F_{k-1} \setminus \{s\}$ under T is again a set of states which does not contain s. Consequently, eliminating s from F_k makes the current approximation "more inductive", increasing the chance to satisfy condition (b) above and to find an inductive invariant.

If Query (3.3) fails, there exists a CTI t in F_{k-1} (a predecessor of s), and IC3 proceeds to refine F_{k-1} until either no more predecessors are found or the initial states are reached. In the latter case, IC3 has found a (non-spurious) counterexample.

One remarkable characteristic of IC3 is that it avoids the costly unwinding required by interpolation-based model checkers (Section 3.1.1) or BMC (Formula (2.2)).

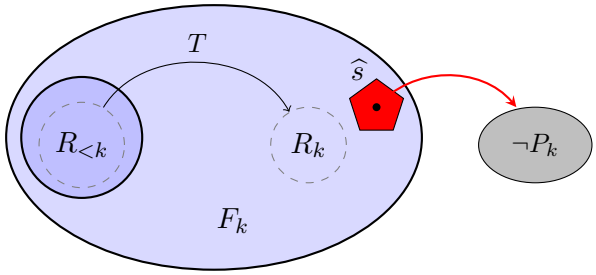


(a) Counterexample to induction

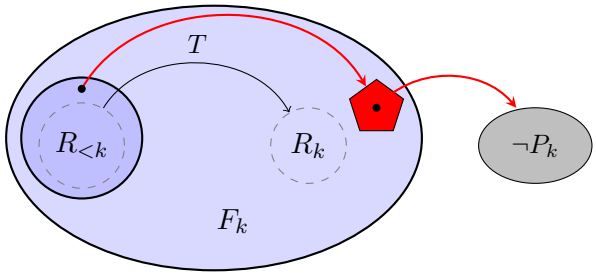


(b) Refined approximation

Figure 3.5: Induction-based refinement in IC3



(a) Abstract CTI



(b) Spurious CTI introduced by abstraction

Figure 3.6: IC3 and predicate abstraction

3.2.2 IC3 for Software

While the basic algorithm from Section 3.2.1 is readily applicable to transition relations T that encode software, the large number of program states severely hampers scalability, since it requires IC3 to deal with a large number of CTIs. Our publication [BBW14] in Chapter 6 addresses this limitation by applying predicate abstraction (cf. Section 2.3) to CTIs. Figure 3.6a shows an abstraction $\widehat{s}$ (in red) of a CTI s. As previously, IC3 attempts to eliminate $\widehat{s}$ using a query similar to (3.3):

$$\neg\widehat{s}(V) \wedge F_{k-1}(V) \wedge T(V, V') \stackrel{?}{\Rightarrow} \neg\widehat{s}(V') \quad (3.4)$$

If the query succeeds, IC3 can eliminate the abstract CTI, which encompasses significantly more states than only s. Consequently, eliminating the abstract CTI constitutes a larger progress towards finding an inductive invariant than eliminating a concrete CTI.

Otherwise, the query yields a predecessor t of $\widehat{s}$. Therefore, $\widehat{s}$ cannot be eliminated since it might be reachable from I (and therefore part of a counterexample). Unfortunately, the CTI t could be an artifact of applying predicate abstraction (illustrated in Figure 3.6b), i.e., t is not a predecessor of the original CTI s. If this is the case, predicate abstraction effectively thwarted refinement of the approximation F_k.

Proceeding with CTI t might eventually lead to a spurious counterexample which can be used as a trigger to refine the abstract CTI $\widehat{s}$. Unfortunately, this requires IC3 to generate a full-length counterexample, delaying the refinement of F_k significantly. Our paper [BBW14] in Chapter 6 avoids this problem by introducing a novel

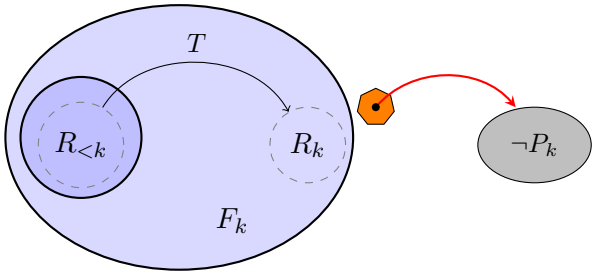


Figure 3.7: Refined CTI and approximation

refinement mechanism that allows us to refine $\widehat{s}$ instantly. Moreover, the refinement mechanism relies on interpolation and extracts refinement predicates from a single transition T (from Query (3.3), in fact) rather than a full-length counterexample. The result of refining $\widehat{s}$ and F_k this way is illustrated in Figure 3.7.

The novel abstraction-refinement scheme presented in [BBW14] (Chapter 6) improves over existing counterexample-based refinement techniques in the following ways:

1. Refinement of abstract CTIs is immediate and does not require IC3 to generate a full-length counterexample.

2. At the same time, refinement can be delayed up to the point that a counterexample is found, allowing the algorithm to trigger refinement at any point in time. An experimental evaluation of different strategies is provided.

3. Refinement predicates are extracted by means of interpolation from a formula encoding a single step of the transition relation, thus avoiding a costly unwinding of T.

Our experimental evaluation shows that our implementation solves more of the benchmark instances presented in [BBW14] in Chapter 6 than CPAchecker [BHT07], the verification tool that won the 2015 Software Verification Competition (SV-COMP) [Bey15]. A reimplementation of our approach targeting multi-threaded software [GLW16] was also submitted to SV-COMP 2016. In the concurrency category, our IC3-based verifier was only outperformed by incomplete BMC-based verification tools (which are unable to conclusively prove safety), and performed significantly better than other complete interpolation-based model checkers such as Impara [WKO13].

3.3 Checking for Bugs Faster

Bugs whose detection hinges on extremely long executions of the system under test—such as the arithmetic overflow in the 787 Dreamliner, which surfaced only after 248 days of continuous operation—are particularly challenging for verification tools. Property violations which require many iterations of loops require particularly deep probing of the state space.

While the acceleration techniques presented in Section 2.2 can theoretically remedy this problem, their practical value is limited, as linear arithmetic does not accurately model real software. Linear arithmetic does not take overflows into account: the formula $(i + 1 < i)$ is unsatisfiable if i takes values in $\mathbb{N}$, but satisfiable if i is an unsigned 32-bit variable with $0 \leq i < 2^{32}$ (since $i + 1 = 0$ if $i = 2^{32} - 1$). Consequently, acceleration based on the assumption that i is unbounded is unsound. An inaccurate model of bit-vector arithmetic may result in missed counterexamples as well as false positives (i.e., unjustified warnings). Ultimately, a verification approach using the natural numbers $\mathbb{N}$ to model bit-vectors cannot be trusted.

```
void* memset  ( void *buf, int c,
                size_t len  )
{
    for(size_t i=0; i<len; i++)
        ((char*)buf)[i]=c;
    return buf;
}
```

Figure 3.8: Code fragment updating an array

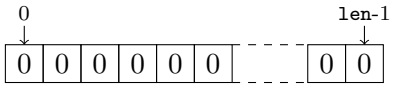


Figure 3.9: Effect of executing `memset`

The situation is even worse if the program under test makes use of arrays, such as the program fragment in Figure 3.8. The effect of executing the instruction `memset(buf, 0, len)` is shown in Figure 3.9: the first `len` elements of the array `buf` are zeroed out. As this effect cannot be modeled in linear arithmetic, the transitive closure of a transition T modeling `memset(buf, 0, len)` cannot be computed using the technique of Section 2.2. This shortcoming is particularly problematic as the verification of programs with buffer overflows—a bug that is encountered frequently—would particularly benefit from acceleration.

3.3.1 Bit-Vectors and Arrays

Our journal paper [KLW15b] in Chapter 7 introduces a novel acceleration technique that enables us to deal with programs with bit-vectors as well as arrays (under certain limitations). The instruction `i++` highlighted in Figure 3.8 is modeled as $\mathtt{i}' = \mathtt{i} + 1$ (as in Figure 2.6b). If the resulting transition occurs in the body of a loop, it can be reformulated as a recurrence equation $\mathtt{i}_n = \mathtt{i}_{n-1} + 1$ with $\mathtt{i}_0 = \mathtt{i}$ (an inductive definition of $\mathtt{i}_n$), whose closed form $\mathtt{i}_n = \mathtt{i}_0 + n$ can be obtained using a constraint solver. This closed form corresponds to an accelerated version of the original transition – if we assume that `i` is unbounded.

To obtain an accelerated transition that does not ignore the effects of arithmetic overflows, we need to restrict n to the range in which linear arithmetic accurately models bit-vector arithmetic. In other words, the acceleration is "stopped" just before an arithmetic overflow occurs:

$$\exists n \leq (\mathtt{INT\_MAX} - \mathtt{i}) \,.\, \mathtt{i}' = \mathtt{i} + n \tag{3.5}$$

The resulting transition $T^{\langle n \rangle}$ represents an *under*-approximation of the transitive closure T^*, and can in general not be used to determine *all* reachable states in one step. Consequently, model checking tools still need to consider multiple iterations of $T^{\langle n \rangle}$ when verifying the system. The number of required iterations to find a fixpoint or a bug, however, is still reduced significantly in comparison to the original transition relation T.

We accelerate the array assignment in Figure 3.8 using McCarthy's theory of arrays [McC93] to formalize the effect of executing `buf[i]=c` n times:

$$\begin{pmatrix} \forall j \leq n \,.\, \mathtt{buf}'[\mathtt{i}+j] = c \quad \wedge \\ \forall j > n \,.\, \mathtt{buf}'[\mathtt{i}+j] = \mathtt{buf}[\mathtt{i}+j] \end{pmatrix} \tag{3.6}$$

The primed counterpart $\mathtt{buf}'$ of `buf` is constrained to be c within the range determined by Equation (3.5), which is achieved by deploying a universal quantifier in Formula (3.6). Unfortunately, quantified transition relations complicate model checking significantly. While SMT-based BMC can still be used to find deep

bugs, interpolation-based program verification is severely limited by the lack of interpolating decision procedures for quantified formulas. To limit this undesirable side-effect, Chapter 7 provides heuristics that help reduce the number of quantifiers in accelerated transition relations.

3.3.2 Ruling Out Bugs Faster

In our paper [KLW15a] in Chapter 8 we present an approach that avoids the challenge of computing fixpoints in the presence of quantifiers by falling back on BMC (cf. Equation (2.2) in Section 2.1). The presence of bugs can be determined efficiently using a BMC instance that incorporates the under-approximation $T^{\langle n \rangle}$ of T^*:

$$I(V_0) \wedge \left(\bigwedge_{i=1}^{k} T^{\langle n \rangle}(V_{i-1}, V_i) \right) \wedge \neg P(V_k) \quad (3.7)$$

Despite the fact that this approach allows us to explore a vastly larger portion of the state space than traditional BMC, it inherits the limitation that the existence of bugs cannot be ruled out definitely.

In certain cases, however, the correctness of a system can still be established by means of BMC, namely if it can be shown that increasing k in Equations (2.2) or (3.7) does not result in additional execution traces [CKOS04, CKL04]. Intuitively, this applies if all loops in T have been explored exhaustively and the longest execution of the system has at most k steps. While this bound is rarely reached in traditional BMC, deploying the under-approximation $T^{\langle n \rangle}$ of T^* in Equation (3.7) allows us to reach this threshold much quicker. Our paper [KLW15a] in Chapter 8 explains how to instantiate the threshold-detection techniques introduced in [CKOS04, CKL04] in the presence of accelerated transition relations.

The contributions of Chapters 7 and 8 ([KLW15b] and [KLW15a], respectively) can be summarized as follows:

1. We introduce the notion of loop under-approximation, which allows us to perform limited acceleration of transition relations in the presence of bit-vectors and arrays.

2. We demonstrate that under-approximation enables an exponential speed-up in the detection of a wide range of buffer overflow bugs collected in [KHCL07].

3. Our novel threshold-based verification techniques allow for the BMC-based verification of accelerated transition relations, enabling us to verify programs with arrays that are out of reach even for interpolation-based software model checkers.

3.4 Silicon Fault Localization

Owing to the high cost of recalling and replacing faulty circuits, verification and validation has a particularly high significance in hardware design [CGP99]. Chip manufacturers were the first to adopt model checking in their development process (as stressed in Chapter 4). The model checking algorithms listed in Chapter 4 can be readily applied to hardware designs (provided in hardware description languages such as VHDL or Verilog as in Figure 3.10) or logic net-lists as in Figure 2.7), ensuring the correctness of the design at early "pre-silicon" development stages.

Functional correctness of the high-level hardware design, however, does not guarantee the absence of bugs in the chip prototype or the final integrated circuit. Electrical faults introduced during the manufacturing process are not reflected by the high-level model and need to be

```
1:  always@(posedge clk)
2:    if (ue[1]) begin
3:      IP = IP + len;
4:      if (btaken)
5:        IP = IP + dist;
6:    end
```

Figure 3.10: Verilog code

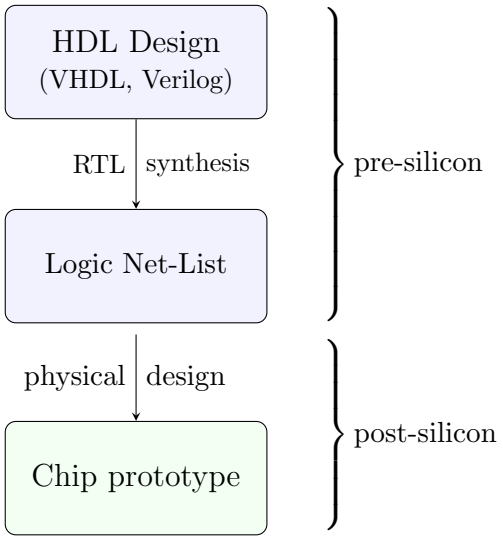


Figure 3.11: Hardware development

caught in the "post-silicon" state of hardware development (Figure 3.11). The "Rowhammer" security vulnerability described in Chapter 1 is a prominent example of a bug introduced during the manufacturing process. The vulnerability is a result of the increasing density of integrated circuits; the physical proximity of individual DRAM cells results in an undesired correlation between signals (so-called bridging faults). Consequently, signals in one row of the DRAM circuit influence cells in adjacent rows, allowing

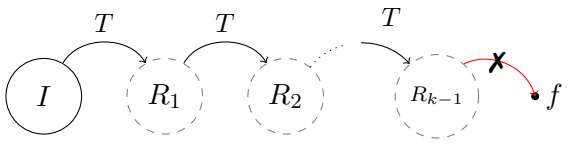


Figure 3.12: Discrepancy between model and reality

the attacker to draw conclusions about regions of the memory that should remain inaccessible.

The cost of post-silicon validation is high: 35 percent of the development cycle of a new chip being are spent on debugging hardware prototypes [ABD+06]. The fact that test scenarios can be executed at full speed (unlike in model checking or simulation of the high-level model) allows for extensive testing of the prototype and can result in extremely lengthy erroneous execution traces. Locating faults in such a trace is particularly challenging due to limited observability of signals in hardware. Only a small percentage of the state space traversed by an erroneous execution can be recorded using trace buffers and scan chains.

3.4.1 Interpolation-based Diagnosis

Our paper [ZWM14] in Chapter 9 resorts to consistency-based diagnosis [Rei87] to locate the temporal and spatial location of electrical faults in an execution. Given a crash state f obtained by running the faulty integrated circuit, the approach identifies gates in the high-level design whose malfunction may have caused the observed erroneous behavior. The logic net-list is presumed to be a correct ("golden") model and therefore f is not reachable via the transition relation T (see Figure 3.12). Consequently, an un-

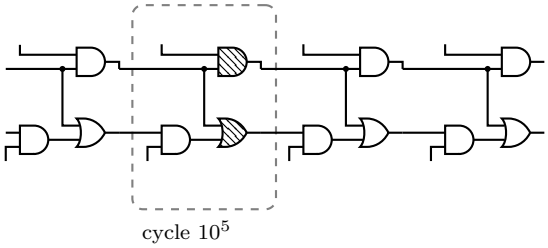


Figure 3.13: Cycle-based fault localization

winding of T constrained with the initial state of the failing test scenario and the final failure state f is necessarily unsatisfiable:

$$I(V_0) \wedge \left(\bigwedge_{i=1}^{k} T(V_{i-1}, V_i) \right) \wedge f(V_k) \qquad (3.8)$$

Figure 3.13 shows part of such a multi-cycle unfolding of the net-list in Figure 2.7. Similar to proof-based abstraction (Section 2.3), a (minimal) unsatisfiable core obtained from Formula (3.8) encompasses the encoding of all gates and signals relevant to the failure (e.g., the highlighted gates in Figure 3.13).

The challenge is that k (the number of cycles executed before a failing state is reached) can be extremely large in the context of post-silicon fault localization. Consequently, the computational effort to compute a core of Formula (3.8) can be prohibitively large.

Chapter 9 ([ZWM14]) addresses this challenge by using an interpolant sequence $\mathrm{Itp}_1, \ldots, \mathrm{Itp}_k$ (as in Figure 3.3) to split Formula (3.8) such that for every i with $1 < i \leq k$ we have

$$\mathrm{Itp}_{i-1}(V_{i-1}) \wedge T(V_{i-1}, V_i) \Rightarrow \mathrm{Itp}_i(V_i)\,. \qquad (3.9)$$

Intuitively, each Itp_i represents safe states of the circuit. Since $f \notin \mathrm{Itp}_k$ by construction, the faulty execution of the integrated circuit must have diverged from Itp_i at some cycle $i-1$. Consequently, candidates for faulty gates in cycle $i-1$ can be derived from an unsatisfiable core of

$$\mathrm{Itp}_{i-1}(V_{i-1}) \wedge T(V_{i-1}, V_i) \wedge \neg\, \mathrm{Itp}_i(V_i)\,, \qquad (3.10)$$

allowing us to focus our localization efforts on one cycle of the execution at a time (e.g., cycle 10^5 in Figure 3.13). As a result, our approach is able to determine candidates for faulty gates as well as for the cycle in which the fault occurred.

3.4.2 Interpolants without Proofs

Unlike in Section 3.1.2, it is not feasible to extract the interpolants from a refutation proof of Formula (3.8), since the solver cannot construct such a proof without considering Formula (3.8) in its entirety. To avoid this problem, we deploy a different approach to compute interpolants in [ZWM14] in Chapter 9. Starting from the final failure state f of the circuit, we use the transition relation T to propagate information backwards. First, we take $\mathrm{Itp}_k(V_k)$ to be $\neg f(V_k)$. Then, we use Itp_i (and the partial state information obtained from trace buffers, which is omitted in the following formula) to find an interpolant Itp_{i-1} satisfying the following condition:

$$\neg\, \mathrm{Itp}_i(V_i) \wedge T(V_{i-1}, V_i) \Rightarrow \neg\, \mathrm{Itp}_{i-1}(V_{i-1}) \quad (3.11)$$

Itp_{i-1} is constructed using the *backbone* of the formula $\mathrm{Itp}_i(V_i) \wedge T(V_{i-1}, V_i)$ [ZWM11], a consequence of logical $\mathrm{Itp}_i(V_i) \wedge T(V_{i-1}, V_i)$ which can be efficiently computed using a SAT solver [ZWSM11]. The propagation stops as soon as an inconsistency with T and the partially observed state is encountered, or if the initial cycle is reached.

While the information obtained via backbones is not necessarily sufficient to construct an interpolant sequence, a strategic selection of the

signals of the circuit tracked by trace buffers significantly increases the chances to succeed [ZWM12].

In summary, the contributions in Chapter 9 (our paper [ZWM14], respectively) include:

1. A novel framework for interpolation-based fault diagnosis, which achieves scalability by focusing the localization effort on single cycles.

2. A novel approach to extract interpolant sequences from failed executions by using backbones to derive logical consequences of formulas.

3. An evaluation of our localization technique on a range of circuits including the microcontroller designs 86HC05 and 8051.

3.5 Concluding Comments

Interpolation and induction—the central theme of this habilitation thesis—are essential techniques in automated verification that have wide-ranging applications. This thesis summarizes my effort to advance the field of interpolation- and induction-based verification techniques on a theoretical as well as practical level. The following (independent) chapters are peer-reviewed publications documenting this effort.

Bibliography

[ABD+06] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Design Automation Conference (DAC)*, pages 7–12. ACM, 2006.

[BBW14] Johannes Birgmeier, Aaron Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 829–846. Springer, 2014.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Verification (IFM)*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science (LICS)*, pages 428–439. IEEE, 1990.

[Bey15] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Boi99] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1999.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL92] E. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Principles of Programming Languages (POPL)*, pages 343–354. ACM, 1992.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *Verification, Model Checking and*

Abstract Interpretation (VMCAI), volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[FL02] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[Fos12] Harry Foster. The 2012 Wilson Research Group functional verification study. Available at `https://blogs.mentor.com/verificationhorizons/blog/2011/03/30/prologue-the-2010-wilson-research-group-functional-verification-study/` (last accessed: December 7, 2015), 2012.

[Gib15] Samuel Gibbs. US aviation authority: Boeing 787 bug could cause 'loss of control'. *The Guardian*, May 1, 2015. Available at `http://gu.com/p/48333/sbl` (last accessed: December 3, 2015).

[GLW16] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: Parallel software with IC3 – (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer, 2016.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HBS13] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Better generalization in IC3. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 157–164. FMCAD Inc., 2013.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002.

[Jha04] Ranjit Jhala. *Program Verification by Lazy Abstraction*. PhD thesis, University of California Berkeley, 2004.

[Joh14] Chris Johnston. Nats: computer failure behind London airport chaos was unprecedented. *The Guardian*, December 13, 2014. Available at `http://gu.com/p/446gb/sbl` (last accessed: December 3, 2015).

[Kel14] Heather Kelly. The 'Heartbleed' security flaw that affects most of the Internet. Available at `http://edition.cnn.com/2014/04/08/tech/web/heartbleed-openssl/index.html` (last accessed: December 3, 2015), April 9, 2014.

[KHCL07] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Automated Software Engineering (ASE)*, pages 389–392. ACM, 2007.

[Kin70] James C. King. *A program verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

[Kir15] Jeremy Kirk. Researchers develop astonishing web-based attack on a computer's DRAM. Available at `http://www.infoworld.com/article/2954722/security/researchers-develop-astonishing-webbased-attack-on-a-computers-dram.html` (last accessed: December 3, 2015), July 30, 2015.

[KLW13] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2013.

[KLW15a] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Proving safety with trace automata and bounded model checking. In *Formal Methods (FM)*, volume 9109 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2015.

[KLW15b] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. *Formal Methods in Systems Design (FMSD)*, 47(1):75–92, 2015.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: An algorithmic point of view*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[KW06] Daniel Kroening and Georg Weissenbacher. Counterexamples with loops for predicate abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 152–165. Springer, 2006.

[KW10] Daniel Kroening and Georg Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Aspects of Computing*, 22(2):105–128, 2010.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with wolverine. In *Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 573–578. Springer, 2011.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[McC93] John McCarthy. Towards a mathematical science of computation. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification: Fundamental Issues in Computer Science*, pages 35–56. Springer, 1993.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[McM06a] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[McM06b] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011.

[MW12] Sharad Malik and Georg Weissenbacher. Boolean satisfiability solvers: techniques and extensions. In *Software Safety and Security - Tools for Analysis and Verification*, NATO Science for Peace and Security Series. IOS Press, 2012.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic (JSL)*, 62(3):981–998, 1997.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351, 1982.

[Rei87] Ray Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[SW16] Matthias Schlaipfer and Georg Weissenbacher. Labelled interpolation systems for hyper-resolution, clausal, and local proofs. *Journal of Automated Reasoning (JAR)*, 57(1):3–36, 2016.

[TT14] Hiroko Tabuchi and Jaclyn Trop. Toyota recalls newest Priuses over software. *The New York Times*, page B1, February 13, 2014. Available at `http://nyti.ms/1ffhP2B` (last accessed: December 3, 2015).

[VWM15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, November 2015.

[Wei10] Georg Weissenbacher. *Program Analysis with Interpolants*. PhD thesis, Oxford, 2010.

[Wei12] Georg Weissenbacher. Interpolant strength revisited. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2012.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 210–217. IEEE, 2013.

[ZWM11] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 63–66. IEEE, 2011.

[ZWM12] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Coverage-based trace signal selection for fault localisation in post-silicon validation. In *Haifa Verification Conference (HVC)*, volume 7857 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2012.

[ZWM14] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Silicon fault diagnosis using sequence interpolation with backbones. In *Computer-Aided Design (ICCAD)*, pages 348–355. IEEE, 2014.

[ZWSM11] Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *High Level Design Validation and Test Workshop (HLDVT)*, pages 84–91. IEEE, 2011.

Part II

Publications

4 | Boolean Satisfiability Solvers and Their Applications in Model Checking

Yakir Vizel, Sharad Malik, and Georg Weissenbacher
Proceedings of the IEEE, Volume 3, Issue 11, 2015
http://dx.doi.org/10.1109/JPROC.2015.2455034

Abstract

Boolean Satisfiability (SAT)—the problem of determining whether there exists an assignment satisfying a given Boolean formula—is a fundamental intractable problem in computer science. SAT has many applications in Electronic Design Automation (EDA), notably in synthesis and verification. Consequently, SAT has received much attention from the EDA community, who developed algorithms that have had a significant impact on the performance of SAT solvers. EDA researchers introduced techniques such as conflict-driven clause learning, novel branching heuristics, and efficient unit propagation. These techniques form the basis of all modern SAT solvers. Using these ideas, contemporary SAT solvers can often handle practical instances with millions of variables and constraints.

The continuing advances of SAT solvers are the driving force of modern model checking tools, which are used to check the correctness of hardware designs. Contemporary automated verification techniques such as Bounded Model Checking, proof-based abstraction, interpolation-based model checking, and IC3 have in common that they are all based on SAT solvers and their extensions.

In this paper, we trace the most important contributions made to modern SAT solvers by the EDA community, and discuss applications of SAT in hardware model checking.

5 | Labelled Interpolation Systems for Hyper-Resolution, Clausal and Local Proofs

Matthias Schlaipfer and Georg Weissenbacher
Journal of Automated Reasoning, Volume 57, Issue 1, 2016
http://dx.doi.org/10.1007/s10817-016-9364-6

(extended version of [Wei12])

Abstract

Craig's interpolation theorem has numerous applications in model checking, automated reasoning, and synthesis. There is a variety of interpolation systems which derive interpolants from refutation proofs; these systems are ad-hoc and rigid in the sense that they provide exactly one interpolant for a given proof. In previous work, we introduced a parametrised interpolation system which subsumes existing interpolation methods for propositional resolution proofs and enables the systematic variation of the logical strength and the elimination of non-essential variables in interpolants. In this paper, we generalise this system to propositional hyper-resolution proofs as well as clausal proofs. The latter are generated by contemporary SAT solvers. Finally, we show that, when applied to local (or split) proofs, our extension generalises two existing interpolation systems for first-order logic and relates them in logical strength.

6 | Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)

Johannes Birgmeier, Aaron Bradley, and Georg Weissenbacher
Conference on Computer Aided Verification (CAV), 2014
http://dx.doi.org/10.1007/978-3-319-08867-9_55

Abstract

Typical CEGAR-based verification methods refine the abstract domain based on full counterexample traces. The finite state model checking algorithm IC3 introduced the concept of discovering, generalizing from, and thereby eliminating individual state counterexamples to induction (CTIs). This focus on individual states suggests a simpler abstraction-refinement scheme in which refinements are performed relative to single steps of the transition relation, thus reducing the expense of refinement and eliminating the need for full traces. Interestingly, this change in refinement focus leads to a natural spectrum of refinement options, including when to refine and which type of concrete single-step query to refine relative to. Experiments validate that CTI-focused abstraction refinement, or CTIGAR, is competitive with existing CEGAR-based tools.

7 | Under-approximating Loops in C Programs for Fast Counterexample Detection

Daniel Kroening, Matt Lewis, and Georg Weissenbacher
Formal Methods in Systems Design, Volume 47, Issue 1, 2015
http://dx.doi.org/10.1007/s10703-015-0228-1

(extended version of [KLW13])

Abstract

Many software model checkers only detect counterexamples with deep loops after exploring numerous spurious and increasingly longer counterexamples. We propose a technique that aims at eliminating this weakness by constructing auxiliary paths that represent the effect of a range of loop iterations. Unlike acceleration, which captures the exact effect of arbitrarily many loop iterations, these auxiliary paths may under-approximate the behaviour of the loops. In return, the approximation is sound with respect to the bit-vector semantics of programs.

Our approach supports arbitrary conditions and assignments to arrays in the loop body, but may as a result introduce quantified conditionals. To reduce the resulting performance penalty, we present two quantifier elimination techniques specially geared towards our application.

Loop under-approximation can be combined with a broad range of verification techniques. We paired our techniques with lazy abstraction and bounded model checking, and evaluated the resulting tool on a number of buffer overflow benchmarks, demonstrating its ability to efficiently detect deep counterexamples in C programs that manipulate

8 | Proving Safety with Trace Automata and Bounded Model Checking

Daniel Kroening, Matt Lewis, and Georg Weissenbacher
Symposium on Formal Methods (FM), 2015
http://dx.doi.org/10.1007/978-3-319-19249-9_21

Abstract

Loop under-approximation enriches C programs with additional branches that represent the effect of a (limited) range of loop iterations. While this technique can speed up bug detection significantly, it introduces redundant execution traces which may complicate the verification of the program. This holds particularly true for tools based on Bounded Model Checking, which incorporate simplistic heuristics to determine whether all feasible iterations of a loop have been considered.

We present a technique that uses *trace automata* to eliminate redundant executions after performing loop acceleration. The method reduces the diameter of the program under analysis, which is in certain cases sufficient to allow a safety proof using Bounded Model Checking. Our transformation is precise—it does not introduce false positives, nor does it mask any errors. We have implemented the analysis as a source-to-source transformation, and present experimental results showing the applicability of the technique.

9 | Silicon Fault Diagnosis Using Sequence Interpolation with Backbones

Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik
International Conference on Computer Aided Design (ICCAD), 2014
http://dx.doi.org/10.1109/ICCAD.2014.7001373

Abstract

Silicon fault diagnosis, the process of locating faults in a chip prototype, becomes more challenging and time-consuming with increasing design complexity. Consistency-based fault diagnosis aims at identifying fault candidates for an erroneous execution trace by symbolically checking the consistency between the golden gate-level model and the faulty behavior of the prototype chip.

The scalability of this technique is limited to short executions due to the underlying decision procedure. This problem has previously been addressed by restricting the analysis to a window of fixed size and moving it along the execution trace. In this setting, limited observability results in a loss of precision and potentially missed fault candidates.

We present a novel interpolation-based framework which formalizes the propagation of state information across sliding windows as a satisfiability problem. Our approach provides both spatial and temporal localization for general faults and is not restricted to a specific fault model. Further, our approach can be used to provide more accurate localization for a single permanent fault model. We experimentally demonstrate the efficacy and scalability of this approach by applying it to a variety of benchmarks from multiple suites (OpenCores, ITC99 and HWMCC).