PDR and SMT2 for Formal Verification of Embedded Systems

UNIKASSEL VERSITÄT

BTC embedded systems

Master Thesis

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Master of Science
in Computer Science

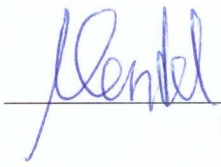
University of Kassel
Department of Electrical Engineering and Computer Science
Faculty of Theoretical Computer Science / Formal Methods

Author:	Lukas Mentel
Examiners:	Prof. Dr. Martin Lange Faculty of Theoretical Computer Science / Formal Methods University of Kassel
	Prof. Dr. Peter Zipf Faculty of Digital Technology University of Kassel
Advisor:	Dr. Karsten Scheibler Principal Research Engineer BTC Embedded Systems AG
Submitted:	December 2, 2024

Hiermit bestätige ich, dass diese Arbeit von mir verfasst wurde und auf meiner eigenen Arbeit basiert, sofern nicht anders angegeben. Keine Arbeit einer anderen Person wurde ohne Quellenangabe in dieser Arbeit verwendet. Alle Referenzen und wörtlichen Auszüge wurden zitiert.

I hereby certify that this work was written by me and is based on my own work, unless otherwise stated. No work of any other person has been used in this work without mentioning the source. All references and verbatim excerpts have been cited.

Kassel, December 2, 2024



Lukas Mentel

Chapter 1

Introduction

IT systems are omnipresent in our daily lives. We use them at work and in our spare time. Their complexity increases continously.

There are many ways to interact with these systems: Some are directly controlled using input devices like a mouse, a keyboard or a touchscreen. Others get their data from some sort of input (a switch, a sensor, ...) and use this data to control other systems (an actuator, a steering column, etc.). These sort of devices are often called *Embedded Systems*. These systems take over more and more safety-critical tasks: They are used in vehicles, medical devices, robots and in many other situations. This means that these systems must be very reliant. Due to their complexity, it is not easy to verify them. This leads to hardware and software bugs: For example, the Pentium-FDIV-Bug was not found before the processors were produced. This lead to high costs and damage to the reputation of Intel.

To check the correctness of safety-critical systems, tests are often used. However, tests can only show the presence of an error, but not the correctness of a system. To do the latter, more sophisticated approaches are needed. This thesis deals with *Model Checking*, a method to investigate every possible execution of a system. Using Model Checking, the correctness of systems can be proven.

This thesis deals with the verification of software systems written in the C programming language. One important class of verification properties that can be checked on these systems is the absence of *dead code* – i.e. unreachable code parts. This is demanded by the ISO 26262 standard [28] which establishes software development standards for the automotive domain.

The commercial tool BTC Embedded*Platform*® is able to perform dead-code detection over C programs. For C-code containing floating-point operations, it currently relies on the SMT solvers CBMC [38] and iSAT3 [36]. Both solvers incorporate *Bounded Model Checking* [8, 16] to show that a code fragment is reachable. For the detection of dead-code, CBMC uses k-Induction while iSAT3 utilizes Craig Interpolation. Furthermore, a prototypical support for the SMT-LIB standard [3] has been added to BTC Embedded*Platform*®. Just like CBMC, it relies on Bounded Model

Checking and k-Induction.

This thesis deals with IC3 [11] and PDR [22] as an alternative approach. They both originate from the domain of Hardware Model Checking. Because of that, the original IC3 paper only considers propositional formulae. Both approaches can be lifted to the theory level which is required to check C programs. To communicate with the solvers, the SMT-LIB format is used. This allows to investigate the performance of several state of the art SMT solvers.

The outline of this thesis is as follows: Chapter 2 contains preliminary information: After discussing traditional Model Checking approaches, the *IC3* and *PDR* algorithms are introduced. Chapter 3 deals with extensions of these algorithms that may help to improve the performance. The experimental evaluation of the mentioned approaches is presented in Chapter 4. In Chapter 5, the essential results of this thesis are summarized and an outlook to possible further work is given.

Chapter 2

Preliminaries

This section deals with the foundational theory of this thesis. After describing the verification tasks and the general concept of Model Checking, the logics and languages used in this thesis are introduced. The section is concluded by a presentation of the Model Checking algorithms that are considered throughout the rest of this document.

2.1 Satisfiability-Based Verification of C Programs

The approaches discussed throughout this thesis deal with formal verification of C programs. This section explains what verification means and shows how to reduce this task to a satisfiability problem.

2.1.1 System under Test (SUT) and Properties

In this section, the systems to be investigated and the properties to be verified on these systems are presented.

The systems to investigate are *embedded systems* written in the C programming languages that fulfill safety-critical tasks. The core of these systems usually follows a fixed scheme: The current state of the system is represented using *state variables*. To control the behavior of the system, *input variables* are used. Finally, the system can communicate information to other systems using *output variables*. The system starts in a given initial state. During the execution of the system, a dedicated module called "step function" is invoked continually. This step function uses the current inputs and the state of the system to determine the next state of the system and to set its output variables. This is everything a step function can do; in particular, there is no user interaction or retrieval of information from, e.g., the file system.

Example 2.1.1. Consider a simple electric window opener. The control buttons as well as the position sensors are inputs to this system. The state contains information about the current moving direction, if any. Using outputs, the window motor is instructed to move the window.

A system aligned to these rules is called *System under Test* (SUT) throughout this thesis.

For the scope of this thesis, the properties to check are *reachability properties*: A particular statement is given, and the task is to check whether that statement can be reached by any combination of inputs and parameters in one or multiple steps. Using that scheme, structural properties (e.g. Statement or Condition coverage) as well as functional properties (requiring specific variables to have a particular value at specific locations or throughout the complete execution) can be formalized. In Section 2.1.6, concrete examples for properties of SUTs will be given.

Note that without loss of generality, it can be assumed that properties refer to the states of the system only: If a property directly incorporates the value of an input or param variable, a new state variable can be added to the system to keep the value of that variable. Output variables refer to input or state variables and can be replaced by their definition. Based on these observations, it is clear that the properties separate the states of the system into two disjoint sets: The states fulfilling the property (called "good" states) and the states violating the property (referred to as "bad" states).

The next section introduces techniques to deal with such systems and properties.

2.1.2 Model Checking

Model Checking [1, 15] is an important technique for formal verification. The system to verify (the SUT in this thesis) is abstracted into a mathematical *model*, which represents important aspects of the system's behavior. The expected properties of the system are expressed in a formal language with precise semantics. Model Checking means to check whether a property holds in *all* possible runs of the system. In the context of this thesis, it must be shown that no bad state is reachable from the initial states of the system. In contrast to normal software testing, where only some execution paths are considered, Model Checking is able to prove not only the presence, but also the absence of errors in the system. If a spurious behavior of the system is detected, Model Checking approaches often retrieve a single flawn execution path, called a *counterexample*, allowing to debug and fix the incorrect behavior.

For simple models and properties, the problem space is finite, which in principle allows an exhaustive examination. However, this is infeasible for practical applications, because the problem space in general grows at least exponentially with the number of state variables. *Symbolic Model Checking* [17] avoids this by applying logical reasoning to consider multiple execution paths at once. The problem still has exponential runtime, but in practice, the needed resources and execution time may be reduced to an acceptable amount.

Example 2.1.2. Consider the SUT given by the following simple graph:

The nodes of the graph are the states of the system. The states are given a symbolic name. The edges of the graph define the possible transitions between the states. State $s^{(0)}$ is the initial state of the system.

The property "$s^{(3)}$ is unreachable" is fulfilled. The property "$s^{(6)}$ is unreachable" is violated, the path $s^{(0)} \to s^{(2)} \to s^{(4)} \to s^{(6)}$ is a counterexample.

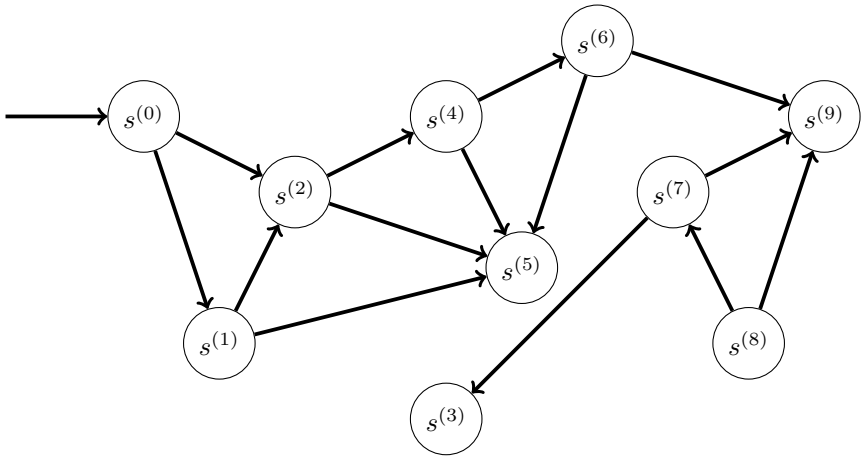


Figure 2.1: A simple SUT

The next sections introduce languages to specify the system and the properties of a given SUT.

2.1.3 Propositional Logic and SAT

Consider the statement "The Burj Khalifa is higher than the Eiffel Tower". It is clear that this statement is either true or false, because the term "larger" only has two different results: Either, the height of the Burj Khalifa is greater than that of the Eiffel Tower, or it is not. *Propositional Logic* is able to express statements about boolean (two-value) domains. *Propositional Formulae* consist of the boolean values *true* and *false*, as well as placeholders for boolean values called *propositional variables*. These so-called *atomic formulae* are connected by boolean operators. In order to evaluate such a formula, concrete values for the variables must be given.

Definition 2.1.3. The set of propositional formulae P_V over a set of propositional variables V are defined as follows:

- *true* and *false* are propositional formulae
- All variables v in V are propositional formulae.
- For propositional formalae φ and ψ, $\neg\varphi$ and $\varphi \wedge \psi$ are propositional formulae.

A *variable assignment* is a mapping $I : V \to \{true, false\}$. I can be extended to an *interpretation* of all propositional formulae P_V as follows:

- $I(true) = true$
- $I(false) = false$
- $I(\neg\varphi) = true$ iff $I(\varphi) = false$

- $I(\varphi \wedge \psi) = \mathit{true}$ iff $I(\varphi) = \mathit{true}$ and $I(\psi) = \mathit{true}$

$\neg(\neg\varphi \wedge \neg\psi)$ is abbreviated as $\varphi \vee \psi$.

Example 2.1.4. $\varphi = A \wedge (\neg B)$ over $V = \{A, B\}$ is a propositional formula. Using the variable assignment $I(A) = \mathit{true}, I(B) = \mathit{false}$, φ evaluates to *true*.

We see that there is no single truth value of a formula; instead, the truth value depends on the interpretation.

For Model Checking, however, the *satisfiability problem (SAT)* is of greater interest: A formula is called *satisfiable* if it has at least one interpretation under which it evalutes to *true*; any such interpretation is then called a *model* of the formula. Here, no specific interpretation is given; satisfiability is a statement about the possible interpretations of a formula.

Example 2.1.5. The formula from Example 2.1.4 is satisfiable. In contrast, the formula $A \wedge \neg A$ is unsatisfiable, since there is no interpretation which can make A and $\neg A$ evaluate to *true* at the same time.

SAT problems are typically solved by techniques like *DPLL* [21] and *Conflict-Driven Clause Learning (CDCL)* [42]. Although SAT is *NP-complete* [18], tools exist that solve reasonable large formulae in an acceptable amount of time [23, 43, 9, 10].

The next section introduces an extension of SAT that allows us to use variables with a non-boolean domain.

2.1.4 Satisfiability Modulo Theories (SMT)

In many situations, statements are established about non-boolean domains. Assigning propositional variables to each atomic statement, as done in the proposition about the Burj Khalifa, often leads to wrong results. An example is the statement "x is greater than y, y is greater than z and z is greater than x". Introducing propositional variables a, b, c for the three atomic statements "x is greater than y", "y is greater than z" "z is greater than x" yields the formula $a \wedge b \wedge c$, which is satisfiable. The original statement, however, is unsatisfiable if x, y, z are real numbers.

Satisfiability Modulo Theories [5] extends propositional logic by so-called *theories*. A theory consists of a collection of values together with operations defined on these values. Multiple theories are combined into *logics*, allowing to define operations about entities from different theories. SMT formulae have the same boolean structure as propositional formulae, but instead of boolean variables and constants, *theory atoms* (expressions from the given logic with a boolean result type) can be used. Terms like variable assignment, interpretation and satisfiability transfer to SMT in the expected way.

Example 2.1.6. The theory of real arithmetic allows to express statements like $x > 42 \vee x + y = z/3$. This formula is satisfied by the variable assignment $x = 127, y = 73, z = 14$. If combined with the theory of the integers, entities from both theories may be used; with i being an integer

and f being a real number, the statement $i = 2 \pmod 5 \wedge x < \pi$ can be expressed; this formula is satisfiable.

As for SAT, there exist tools to check the satisfiability of an SMT formula [29]. Note, however, that SMT theories may have an infinite domain. For example, it is possible to establish statements about Peano arithmetic. This theory is undecidable, because it can express the halting problem of a Turing machine [45]. Therefore, the satisfiability problem for SMT is also undecidable in general.

For finite theories, an approach called *bit-blasting* or *Eager SMT* can be used: The SMT formula is transformed to a SAT formula in such a way that both formulae are at least equi-satisfiable (i.e., the first formula is satisfiable iff the second one is). In many cases, it is even possible to retrieve a model for the SMT formula using a model for the SAT formula.

Another important family of approaches is referred to with the term *Lazy SMT*: The theory atoms are replaced by propositional variables, which produces a propositional formula. If that formula is unsatisfiable, the original SMT formula must be unsatisfiable as well. Models of the SAT formula may lead to spurious models of the SMT formula, in which case the SAT formula is refined to exclude the spurious cases and then checked again. Because Lazy SMT does not push all knowledge to a SAT formula, it can also be applied to theories with an infinite domain [39].

The SMT-LIB standard, which is treated in the next section, allows SMT problems of many different theories to be expressed.

2.1.5 SMT-LIB Standard

Many solvers for different SMT theories were created [29] that use different input languages. To make usage of different SMT solvers more convenient the SMT-LIB initiative was founded [3], which specifies the SMT-LIB standard, currently in version 2.6. This standard specifies a Core theory [44] for propositional formulae and the boolean skeleton of an SMT formula. Furthermore, it contains definitions of SMT theories and logics for different domains [4]. Logics may also introduce quantifiers like *forall* and *exist*.

Example 2.1.7. The propositional formula from Example 2.1.4 can be expressed and checked in the SMT-LIB core theory as follows:

```
(set-option :produce-models true)
(declare-fun A () Bool)
(declare-fun B () Bool)
(declare-fun phi () Bool)

(assert (= phi (and A (not B))))
(check-sat)
(get-model)
(exit)
```

For the scope of this thesis, the quantifier-free logic of bitvectors and floating-points (QF_BVFP) is of interest, because it is able to represent the SUTs as well as the properties to be checked.

Example 2.1.8. For a 32-bit signed integer i and a 32-bit floating-point f, the formula $i = 23 \wedge f = 42.0$ has the following form in SMT-LIB:

```
(set-logic QF_BVFP)
(set-option :produce-models true)

(declare-fun i () (_ BitVec 32))
(declare-fun f () Float32)
(assert (= i #x00000017))
(assert (= f (_ to_fp 8 24) #x42280000))
(check-sat)
(get-model)
(exit)
```

The next section describes the transformation of the SUT and the property in detail.

2.1.6 From C Programs to SMT-LIB Scripts

Safety-critical embedded systems as defined in Section 2.1.1 are often written in C. Such programs cannot be translated directly to SMT-LIB. The reason is that C programs describe a step-by-step execution of a program, whereas SMT-LIB defines a set of logic constraints that do not contain any dynamic execution behavior.

The BTC Embedded*Platform*® allows to perform the necessary modifications. It starts with an SUT as described in Section 2.1.1, i.e. a C program containing a step function that works on input, state, and output variables.

At first, the C code is translated to an intermediate format called SMI. SMI is an imperative programming language (like C) and has similar control flow structures (if, while, switch, ...). An exception is the goto statement which must be expressed using other control structures. During this conversion, *SMI addresses* are introduced for each variable, function, parameter and return value. Pointers to such elements are replaced by `size_t` variables containing the SMI address of the referenced element. The transition needs to distinguish between the "old" state of the SUT, which is evaluated, and the "new" state, which is created. For that, SMI allows to annotate state variables with the step they refer to (previous / next step). The property is injected into the program using a dedicated variable `invariance_property` whose value is 1 initially and switches to 0 if the property is violated. The functions in the program are then *inlined*: the call to the function is replaced by its body. Loops are unrolled, i.e. the loop is replaced by a series of `if` statements each containing one iteration of the loop. If the maximum number of loop iterations can be determined from the source code directly, the loop is unrolled the respective number of times;

otherwise, the number of unrollings must be set by the user. Finally, the program is transformed into a *static-single assignment form*, which means that every variable is assigned at most once. After these transformations, a single function remains encoding the complete verification task (initialization, transitions in an infinite loop, property). This program can then be translated to SMT-LIB. Because Model Checking requires different combinations of these components to be instantiated, five SMT-LIB files are created:

- the header file as a preamble and for general declarations
- the declaration file for declaration of input / output / state variables
- the initialization file for initialization of state variables
- the transition file for a one-step transition
- the property file for the property to be checked

Example 2.1.9. The translation of the following C program:

```
static int counter = 0;
static int executions = 0;

void f(int x)
{
    if (x != 0) { counter += x; }
    else { counter = 0; }

    executions++;
}
```

together with the following property:

```
|counter < executions|
```

produces essentially the following SMT-LIB code (the complete files can be found in Appendix A):

```
; HEADER
; some function definitions -- omitted here

; DECLARATION
(declare-fun counter$$new () (_ BitVec 32))
(declare-fun executions$$new () (_ BitVec 32))
(declare-fun x$$new () (_ BitVec 32))
(declare-fun invariance_property$$new () Bool)
```

```
; INITIALIZATION
(assert (= counter$$0 #x00000000))
(assert (= executions$$0 #x00000000))
(assert invariance_property$$0)

; TRANSITION
(assert (= counter$$new (ite (distinct x$$new 0)
                        (bvadd counter$$old x$$new) #x00000000)))
(assert (= executions$$new (bvadd executions$$old #x00000001)))
(assert (= invariance_property$$new (bvslt counter$$new executions$$new))))

; PROPERTY
(assert invariance_property$$new)
```

The next sections show how to use this SMT-LIB encoding for Model Checking.

2.2 Traditional Approaches for Satisfiability-Based Verification

This section introduces well-established algorithms that are used for formal verification.

2.2.1 Bounded Model Checking (BMC)

Based on the observation in Section 2.1.1, an SUT and its properties can be represented as a graph: The states of the system are the nodes in the graph and the transitions are represented using directed edges from the source of the transition to the target. Furthermore, the initial state as well as the good and bad states can be marked.

Example 2.2.1. The graph in Figure 2.2 represents a very simple SUT together with a property:

The initial state $s^{(0)}$ is marked by the incoming edge without a source. States colored green are good states; states with a red background color are bad states.

A human is able to see that a bad state is reachable using the path $s^{(0)} \to s^{(2)} \to s^{(4)} \to s^{(6)} \to s^{(9)}$. But a computer is not able to visually inspect the graph. Instead, it has to loop over all paths starting from the initial state and check whether any such path leads to a bad state.

An important algorithm for that purpose is *breadth-first search* (BFS). This algorithm starts in the initial state. At first, it checks all paths of length 0, after that all paths of length 1 and so on. Because the graph is finite, the algorithm can stop after n steps if n is the number of states.

Bounded Model Checking (BMC) [8, 16] uses a *symbolic* breadth-first search approach [32, 19]: For each possible length of a path, it creates a formula that represents any possible path of that

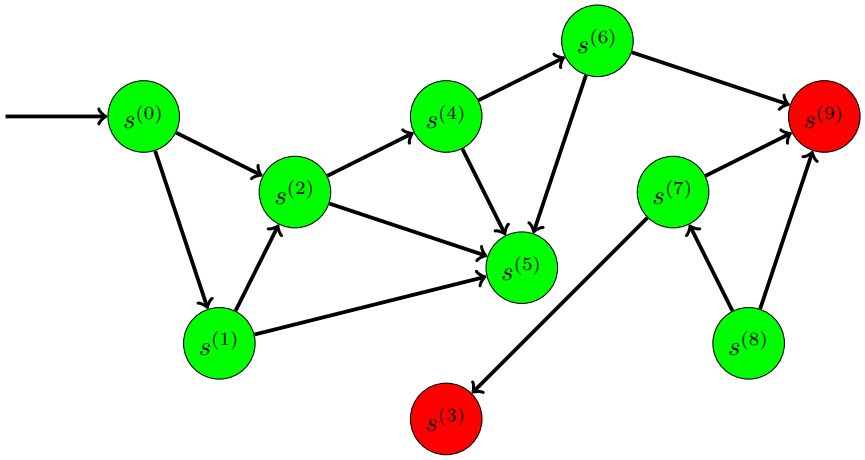


Figure 2.2: A simple SUT

length. In the context of this thesis, SMT formulae are used. The SMT formula for step n is satisfiable iff there is a path of length n starting in the initial and ending in a bad state.

Let I, $T_{n-1,n}$ and P_n be the SMT formulae for the initialization in step 0, the transition from step $n-1$ to step n and the property in step n as shown in Section 2.1.6. Paths of length 0 contain no transitions; the property must be fulfilled by the initial states. The SMT formula encoding that is $\text{BMC}_0 = I \wedge \neg P_0$. Paths of length n contain $n-1$ transitions. Therefore, the formula $\text{BMC}_n = I \wedge T_{0,1} \wedge \cdots \wedge T_{n-1,n} \wedge \neg P_n$ needs to be solved.

Example 2.2.2. For the transition relation in Figure 2.2, the BMC_4 formula is fulfilled by the path $s^{(0)} \to s^{(2)} \to s^{(4)} \to s^{(6)} \to s^{(9)}$: s_0 is the initial state and therefore satisfies I, the transitions between the state are all conformant with T, and the state s_9 is a bad state, thus fulfilling $\neg P_5$.

If all BMC formulae are solved in the natural order, the intermediate steps in the formula are known to fulfill the property, because otherwise, a shorter BMC formula can detect a violation of the property. Therefore, the formula P can be inserted in all previous steps, which leads to the following formula pattern: $I \wedge P_0 \wedge T_{0,1} \wedge P_1 \wedge \cdots \wedge P_{n-1} \wedge T_{n-1,n} \wedge \neg P_n$. This approach is called *extended BMC*.

It is clear that BMC can find paths of any length leading to a bad state in finite systems, because by solving each BMC formula, all paths of any length are considered. This implies that if BMC does not find a spurious path, the correctness of the SUT under the given property is proven, assuming the underlying SMT solver provides correct results. Thus, BMC fulfills our requirements for formal verification. However, the number of steps to investigate can be very large: It is possible to specify properties over SUTs like a binary counter, where the number of steps on the path leading to a bad state is equal to the number of possible SUT states. In order to prove that there is no violation of the property, any paths of any length must always be considered. The next sections deal with approaches to avoid both problems.

2.2.2 k-Induction

Consider the following SUT graph:

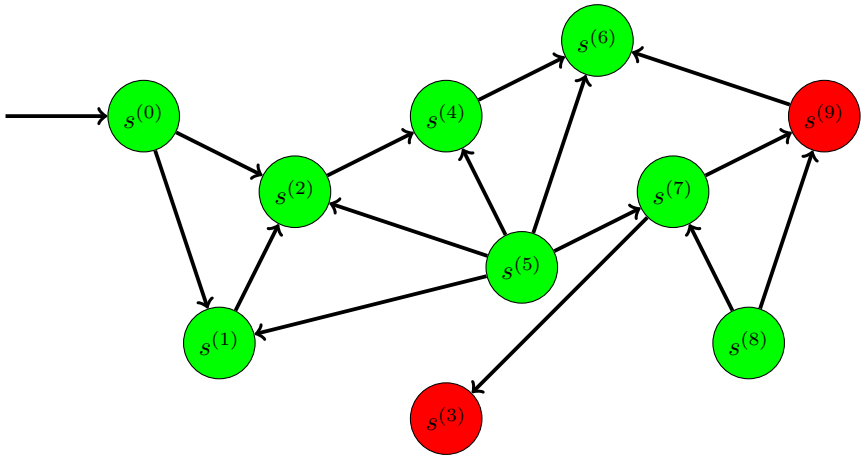


Figure 2.3: A simple SUT

Again, a human can conclude that no bad state is reachable. Computers, on the other hand, need to check that no path leads to a bad state. In particular, it is not sufficient to check that there are no bad states or no transitions from good to bad states: The graph contains a transition from the good state $s^{(7)}$ to the bad state $s^{(9)}$, but the state $s^{(7)}$ is not reachable from the initial state.

Consider the modified BMC formulae from Section 2.2.1:

$$\begin{aligned} \text{BMC}_0 &= I \wedge \neg P_0 \\ \text{BMC}_1 &= I \wedge P_0 \wedge T_{0,1} \wedge \neg P_1 \\ \text{BMC}_2 &= I \wedge P_0 \wedge T_{0,1} \wedge P_1 \wedge T_{1,2} \wedge \neg P_2 \end{aligned} \tag{2.1}$$

An important observation is that formulae with $n \geq 1$ share the common suffix $P_{n-1} \wedge T_{n-1,n} \wedge P_n$. This suffix is satisfiable iff there is a transition from a good to a bad state. The value of n does not impact the satisfiability of that formula, because it is only used in variable names. If this suffix is unsatisfiable, *all* BMC_n formulae with $n \geq 1$ must be unsatisfiable as well, because they contain this suffix in their top-level conjunction. This can also be seen when considering the intuitive meaning of that suffix: If there is no transition from a good to a bad state in the system and the system is initialized in a good state (i.e., BMC_0 does not find a spurious path), then no bad state is reachable, because this would require a transition from a good to a bad state at some point.

For larger BMC formulae, even longer common suffixes can be found: For $n \geq m$, the formulae BMC_n and BMC_m share the suffix $P_{n-m} \wedge T_{n-m,n-m+1} \wedge P_{n-m+1} \wedge \cdots \wedge T_{n-1,n} \wedge \neg P_n$. Intuitively,

this formula encodes that there is a sequence of $m-1$ good states followed by a bad state. Because the value of $n-m$ has no impact on the satisfiability, we can set $n=0$; the resulting formula $\mathrm{IND}_k = P_0 \wedge T_{0,1} \wedge P_1 \wedge \cdots \wedge T_{k-1,k} \wedge \neg P_k$ is the k-Induction formula [24].

Example 2.2.3. For the SUT depicted in Figure 2.3, the BMC_0, BMC_1 and BMC_2 formulae are unsatisfiable, because no bad states are reachable. The IND_1 formula is satisfiable; a satisfying path is $s^{(8)} \to s^{(9)}$: $s^{(8)}$ is a good state whereas $s^{(9)}$ is a bad state. Considering the path $s^{(8)} \to s^{(7)} \to s^{(9)}$, it can be seen that the IND_2 formula is satisfiable as well. The IND_3 formula is unsatisfiable, because there is no path that starts in a good state, passes two other good states and ends in a bad state. This proves that the property cannot be violated.

Like BMC, k-Induction is a symbolic breadth-first search. But there is an important difference between these two approaches: BMC starts at the initial state and tries to find a bad state (forward BFS), whereas k-Induction begins at the bad states, checking if there are paths to these states of a given length (backward BFS).

k-Induction can prove the correctness of the system, but is not able to find counterexamples, because it does not incorporate the initial state of the system. Thus, it is combined with BMC: At first, BMC_0 is checked. Then, for each k, IND_k and BMC_k checks are performed. This algorithm is, like BMC, a decision procedure for the correctness of properties, but it is able to find proofs of correctness much faster than BMC alone, which makes it practically applicable.

One major drawback of that approach, however, is that for complex problems, k-Induction takes a large amount of time, because it needs to (symbolically) consider all paths of length k that end in a bad state. For the verification problems considered in this thesis, there is only a single initial state, whereas the number of bad states is large. Thus, k-Induction is required to explore much many paths than BMC.

In contrast, BMC_k only evaluates the paths of length k that start in the initial states. The next sections introduce approaches that can overcome these weaknesses in some scenarios.

2.3 IC3 and PDR

This section deals with the central algorithms considered in this thesis.

2.3.1 Invariants and Invariant Candidates

Section 2.2.1 and Section 2.2.2 presented BMC and k-Induction, both being breadth-first search approaches. Another important class of algorithms are the *depth-first search* (DFS) techniques [19]. In contrast to BFS, DFS starts with a path of length 0 (the starting node) and extends that path until either the target or a dead-end is reached. If no successor node can be found, *backtracking* is applied, i.e., the traversal moves back to predecessor nodes in the path and checks further outgoing edges.

The approach described in this section employs *backward DFS*: At the beginning, one transition is selected that starts in s and ends in a bad state. Afterwards, a predecessor of the state s is retrieved. This process is repeated until the initial state is reached, in which case a path from the initial state to a bad state was found, or if all predecessors of bad states have been visited.

This approach produces correct results. However, it is not very efficient, because nodes and edges may be visited multiple times. Therefore, DFS algorithms maintain a set V containing the already-visited states. For DFS on explicitly-given graphs, a boolean flag `visited` can be added to each node. For the SUTs considered throughout this thesis, this approach is infeasible, because the number of states is too large.

To understand what is done instead, consider the 1-Induction formula $P_0 \wedge T_{0,1} \wedge \neg P_1$. This formula is unsatisfiable if no bad state is reachable from the set of good states in one transition step. Because IND_1 is solved after BMC_0, we know that P contains the initial state. The IND_1 formula implies that there is no transition that leaves P, i.e. starts in P and ends in $\neg P$. Section 2.2.2 contains examples for properties that are not 1-inductive. The idea is to *strengthen* the property, i.e. to remove "uninteresting" states from it to obtain an 1-inductive set. In the previous section, it was made clear that for the scope of this thesis, Model Checking searches for bad states reachable from the initial state. Therefore, the set to build only needs to contain reachable good states.

Definition 2.3.1. A *1-inductive invariant* is a set of states F fulfilling the following properties:

- F contains the initial state, i.e. $I \wedge \neg F_0$ is unsatisfiable
- All states in F are good states, i.e. $F_0 \wedge \neg P_0$ is unsatisfiable
- F is closed under the transition relation, i.e. $F_0 \wedge T_{0,1} \wedge \neg F_1$ is unsatisfiable

With the previous explanation, it should be clear that a 1-inductive invariant is sufficient to prove that the property holds in the given SUT. Furthermore, if P makes IND_1 unsatisfiable (and BMC_0 is unsatisfiable as well), P itself is a 1-inductive invariant. Because P contains all good states, sets that are 1-inductive must be subsets of P.

Example 2.3.2. Consider the graph in Figure 2.3. The set $\{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(6)}\}$ is a 1-inductive invariant, because it contains $s^{(0)}$, all reachable states, and only good states. The set $P = \{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(5)}, s^{(6)}, s^{(7)}, s^{(8)}\}$ is *not* 1-inductive, because it can be left using the transition $s^{(7)} \to s^{(3)}$.

When looking at general k-Induction, this concept can be extended further:

Definition 2.3.3. A *k-inductive invariant* is a set of states F fulfilling the first two properties as above that makes the k-Induction formula $F_0 \wedge T_{0,1} \wedge F_1 \wedge \cdots \wedge T_{k-1,k} \wedge F_k$ unsatisfiable.

If P makes IND_k unsatisfiable, P is a k-inductive invariant. Furthermore, a set that renders IND_k unsatisfiable does the same for each IND_l with $l \geq k$.

Example 2.3.4. In Figure 2.3, the set $P = \{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(5)}, s^{(6)}, s^{(7)}, s^{(8)}\}$ is a 3-inductive invariant, because there are no paths of length 3 that start in a good state, pass two other good states and end in a bad state. It is also k-inductive for $k \geq 3$.

The goal of the approaches to be presented is to find a k-inductive invariant. However, a computer cannot just look at a picture like a human. Instead, such an invariant is constructed stepwise. During this process, sets are created that are "almost" invariants, but need to be refined further.

Definition 2.3.5. An *invariant candidate* is a set F with the following properties:

- F contains the initial state, i.e. $I \wedge \neg F_0$ is unsatisfiable
- All states in F are good states, i.e. $F_0 \wedge \neg P_0$ is unsatisfiable

Note that invariant candidates are not necessarily k-inductive.

Example 2.3.6. In Figure 2.3, the set $\{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(5)}, s^{(6)}\}$ is an invariant candidate, as it is also a 1-inductive invariant. The set $\{s^{(0)}\}$ is also an invariant candidate, because $s^{(0)}$ is the initial state and is good, although it is not a 1-inductive invariant.

The important idea is to create a sequence $F^0, F^1, \ldots, F^n$ of invariant candidates that grows monotonically, i.e. $F^i \rightarrow F^{i+1}$ for $i < n$. Incrementally, these invariant candidates are extended by further states until one of them is an k-inductive invariant.

Example 2.3.7. For the graph in Figure 2.3, the following monotonic sequence of invariant candidates can be found:

- $F^0 = \{s^{(0)}\}$
- $F^1 = \{s^{(0)}, s^{(1)}, s^{(2)}\}$
- $F^2 = \{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(5)}\}$
- $F^3 = \{s^{(0)}, s^{(1)}, s^{(2)}, s^{(4)}, s^{(5)}, s^{(6)}\}$

The next section introduces the central algorithm of this thesis.

2.3.2 Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3)

Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) is a novel Model Checking approach. It was first published by Aron Bradley [11] and gained much interest in the field of formal method research and application. IC3 incorporates the ideas from Section 2.3.1. This section describes the simplest form of IC3; later sections introduce enhancements.

2.3.3 IC3 Example

Like BMC and k-Induction, IC3 starts with an SUT and a property, given by SMT formulae I, T and P. As a first step, the BMC_0 and BMC_1 formulae are solved, as described in Section 2.2.1.

Example 2.3.8. Consider the graph from Section 2.2.2:

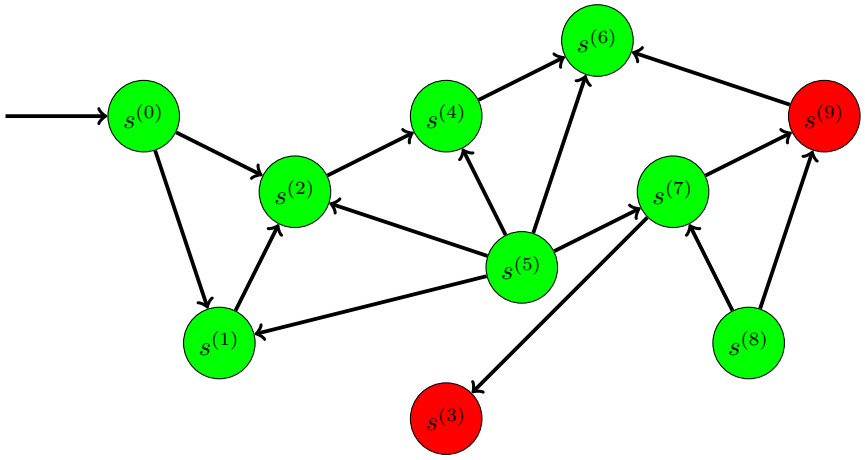


Figure 2.4: A simple SUT

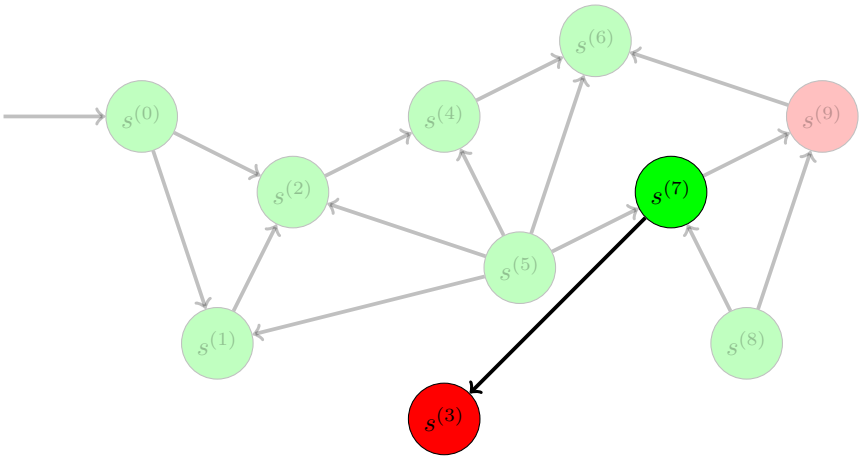
As already mentioned, the BMC_0 and BMC_1 formulae are unsatisfiable for the given SUT and property.

IC3 now creates two sets of states, so called *frames*, denoted by $F^{(i)}$. $F^{(0)}$ is initialized with the initial state; this is the set of states reachable after 0 steps. $F^{(1)}$ contains all good states. This is an over-approximation of the states reachable after 1 step, because the BMC_0 and BMC_1 formulae showed that no bad state is reachable within 0 or 1 steps. The frames are the invariant candidates introduced in Section 2.3.1; the goal is to turn one of them into a 1-inductive invariant.

It needs to be checked whether a bad state is reachable in two or more steps. For that, IC3 selects an arbitrary transition from a state in frame $F^{(1)}$ to a bad state, i.e. a state s from $F^{(1)}$ for which $s_0 \wedge T_{0,1} \rightarrow \neg P_1$. If no such state is present, the algorithm terminates; the property cannot be violated because no transition is possible from any good to any bad state. In IC3, s is called a *proof obligation*. An obligation has the attributes *depth* and *level*. The level describes the number of the frame this obligation is associated with. The depth represents the number of steps to reach a bad state from the obligation.

Example 2.3.9. In Figure 2.4, the transition from $s^{(7)}$ to $s^{(3)}$ has the mentioned characteristics: Therefore, $s^{(7)}$ is a proof obligation. It has level 1 (because it is associated with frame 1) and depth 1 (because one step is needed to reach the bad state $s^{(3)}$ from $s^{(7)}$).

It is now clear that the set of bad states has a predecessor. However, it is unknown whether this predecessor itself is reachable from the initial state.



To investigate that, IC3 now tries to find a predecessor in $F^{(0)}$ for the state s from $F^{(0)}$ i.e. it checks whether a transition is possible from the initial state to s. If that is the case, a counterexample of length 2 is found, because the path starts in the initial state and ends in a bad state. If, however, s has no predecessor in $F^{(0)}$, we know that s is not reachable in 1 step from the initial state. Because the frame $F^{(1)}$ is an over-approximation of the states reachable after 1 step, the state s can be removed from $F^{(1)}$. It is called *blocked cube* for $F^{(1)}$.

Example 2.3.10. In the example SUT, no transition is possible from the initial state $s^{(0)}$ to $s^{(7)}$, so $s^{(7)}$ is turned into a blocked cube for $F^{(1)}$.

If a blocked cube was created, the producing obligation can be *moved* to the next frame. It may be the case that this frame is not created yet. Therefore, obligations have the level attribute, which is incremented by 1 if the obligation is moved.

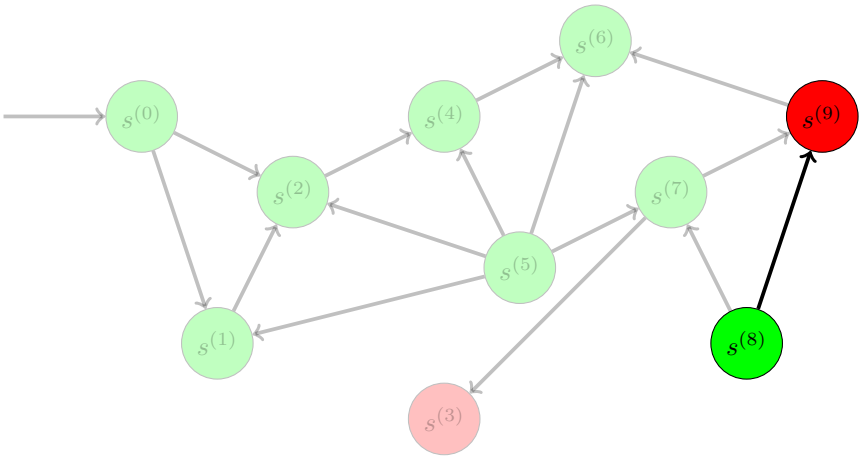
Example 2.3.11. The obligation $s^{(7)}$ is moved when creating the blocked cube. It now has the level 2 and the depth 1.

Because the frame $F^{(2)}$ is not present yet, the obligation cannot be investigated further. Instead, one tries to find a new obligation for frame 1, i.e. a predecessor for the bad state that is not part of the blocked cube at this frame. For that obligation, IC3 again tries to find a predecessor state in $F^{(0)}$. This procedure is repeated until frame 0 is reached or until no further obligation can be created in $F^{(1)}$. This means that all successor states of $F^{(1)}$ are good states.

Example 2.3.12. The bad state $s^{(9)}$ has the predecessor $s^{(8)}$, so $s^{(8)}$ is a proof obligation with level 1 and depth 1.

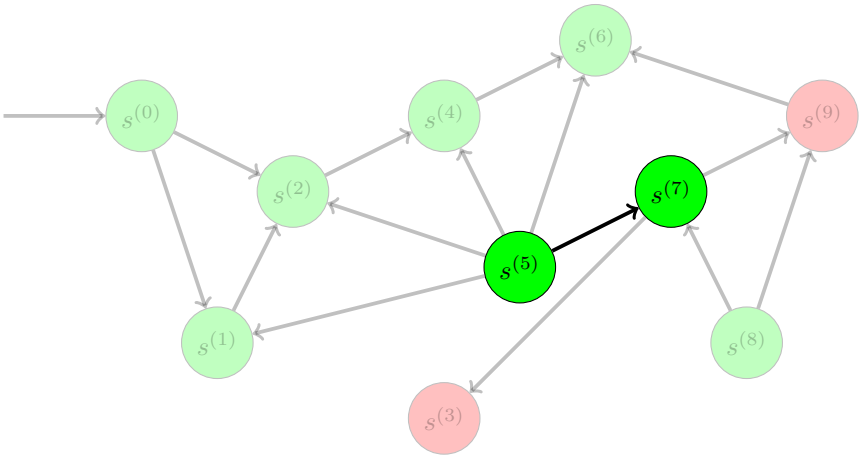
The state $s^{(8)}$ itself is not reachable from $s^{(0)}$; therefore, $s^{(8)}$ is added as blocked cube to $F^{(1)}$ and the obligation is shifted to level 2. After that, no further bad state is reachable from $F^{(1)}$.

It is now clear that no violation of the property can be found within 2 steps from the initial state. But how about more steps?



To answer that question, IC3 creates a new frame $F^{(2)}$. This frame is initialized with the set of good states, just like $F^{(1)}$. To speedup the execution, IC3 now tries to *push* all blocked cubes from $F^{(1)}$ to $F^{(2)}$: It checks for each blocked cube whether it is also unreachable from $F^{(1)}$. If that is the case, the blocked cube is also added to $F^{(2)}$.

Example 2.3.13. The frame $F^{(1)}$ contains the blocked cubes $s^{(7)}$ and $s^{(8)}$.

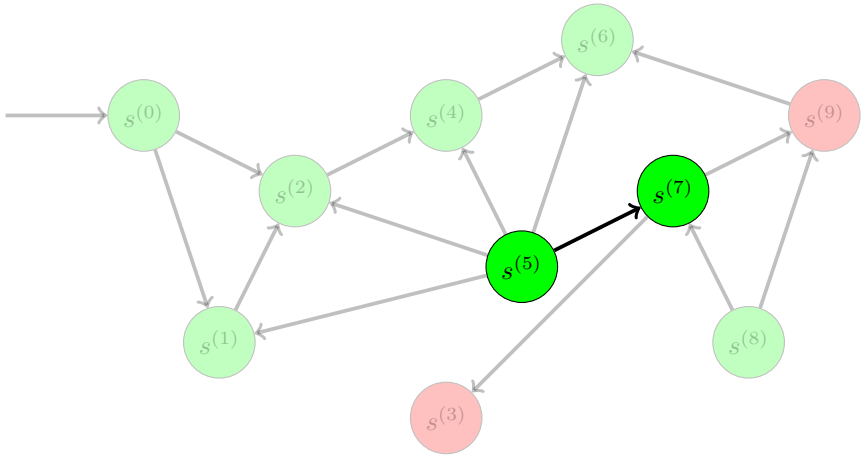


$s^{(7)}$ is reachable from the state $s^{(5)}$ in $F^{(1)}$ and can therefore not be pushed. $s^{(8)}$ has no incoming edges; this means it is not reachable from $F^{(1)}$. The blocked cube $s^{(8)}$ is added to frame $F^{(2)}$. The proof obligation $s^{(8)}$ is moved; it now has level 3 and depth 1.

After that, IC3 now considers the remaining open proof obligations at level 2: As before, it is tried to reach the frame $F^{(0)}$. It may happen that more than one proof obligation is present at a given time. To decide which to handle first, a common heuristic is used: All obligations are stored in a priority queue. The priority of an obligation is higher than the priority of another

obligation if the former one has a smaller level than the latter one. If two obligations have the same level, the obligation with the smaller depth is preferred. If level and depth are equal, their order is determined, e.g., by comparing hash values.

Example 2.3.14. The proof obligation $s^{(7)}$ with level 2 and depth 1 has the predecessor $s^{(5)}$ in $F^{(1)}$.



During the recursion step, the level decreases by one and the depth increases by one, so $s^{(5)}$ is a proof obligation with level 1 and depth 2. The obligation $s^{(7)}$ is still present, which makes sense, because it may have other predecessors in $F^{(1)}$. $s^{(5)}$ is preferred over $s^{(7)}$ because the former one has level 1 and the latter one has level 2. $s^{(5)}$ itself is unreachable from Frame $F^{(0)}$; the obligation is turned into a blocked cube for $F^{(1)}$ and the obligation is moved to level 2. $s^{(5)}$ is also unreachable from frame $F^{(1)}$, so the blocked cube is pushed to $F^{(2)}$ (the obligation is moved to level 3). Pushing the new blocked cube at this point is not necessary for the correctness / completeness of IC3, but often leads to a faster execution in practice. $s^{(7)}$ has only one other incoming edge starting at $s^{(8)}$, which has been excluded from $F^{(1)}$. Thus, a blocked cube for $s^{(7)}$ is created at level 2 and the obligation $s^{(7)}$ is moved to level 3 (it still has depth 1).

If no bad state is reachable from $F^{(2)}$, it is clear that the property cannot be violated within three steps. To investigate longer paths, a new frame is created, pushing all blocked cubes if possible. This procedure is repeated until all blocked cubes could be pushed from a frame $F^{(i)}$ to the frame $F^{(i+1)}$.

Example 2.3.15. The states $s^{(7)}$ and $s^{(8)}$ have been excluded from $F^{(2)}$. Thus, no bad state is reachable from $F^{(2)}$. A new frame $F^{(3)}$ is created and initialized with the property. The blocked cubes $s^{(7)}$ and $s^{(8)}$ are both unreachable from $F^{(2)}$, because $s^{(5)}$, $s^{(7)}$ and $s^{(8)}$ have been excluded from that frame. All blocked cubes are pushed from frame $F^{(2)}$ to frame $F^{(3)}$.

At this point, the sets described by $F^{(i)}$ and $F^{(i+1)}$ are equal, because the same states are blocked in both frames. This also implies that no further bad state is reachable from $F^{(3)}$: If such

a state exists, it would also be reachable from $F^{(i)}$, meaning that the frame $F^{(3)}$ would not be created. Thus, the set of states described by $F^{(i)}$ and $F^{(i+1)}$ is a 1-inductive invariant, as defined in Section 2.3.1.

2.3.4 IC3 algorithm

As for BMC and k-Induction, a symbolic approach is used; for each step described above, SMT formulae are created and their satisfiability is checked using an SMT solver. With this knowledge, the IC3 algorithm can now be given in pseudo-code. Its basic part is the function ic3() that evaluates an SUT and property described by SMT formulae I, T and P. This function returns true if the property is proven and false if a counterexample is found.

```
1  bool ic3():
2    if base_cases() = false:
3      return false
4    F^(0) := I
5    Q := empty priority queue
6    k := 1
7    while true:
8      if strengthen(k, Q) = false:
9        return false
10     k := k+1
11     F^(k) := P
12     if propagate(k) = true:
13       return true
```

The function base_cases() returns false if one of the BMC_0 and BMC_1 formulae is satisfiable and true otherwise:

```
1  bool base_cases():
2    if sat(I ∧ ¬P_0) return false
3    if sat(I ∧ T_0,1 ∧ ¬P_1) return false
4    return true
```

The function strengthen(k), creates all possible obligations for frame $F^{(k)}$, tries to recursively extend them until frame 0 is reached, and creates blocked cubes if necessary. It returns false if a counterexample is found and true otherwise:

```
1  bool strengthen(k, Q):
2    while sat(F_0^(k) ∧ T_0,1 ∧ ¬P_1):
3      s := the predecessor state extracted from the witness
4      o := Obligation(s, k, 1)
5      Q.enqueue(s, k, 1)
```

```
6       if resolve_obligations(k, Q) = false:
7         return false
8     return true
```

The function resolve_obligations(k, Q) is used to recursively find predecessors for all existing proof obligations in the priority queue Q and to create blocked cubes if necessary. Like strengthen, it returns false if a counterexample is found and true otherwise.

Note that the formula to be solved also contains $\neg s_0$: The reason is that the reachability of the proof obligation is not impacted by a transition from s to itself. Thus, $\neg s_0$ can be added safely to the formula.

```
1   bool resolve_obligations(k, Q):
2     while Q contains obligation with d < k:
3       (s, l, d) := Q.dequeue()
4       if sat(F_0^(l-1) ∧ ¬s_0 ∧ T_0,1 ∧ s_1):
5         s' := the predecessor state extracted from the witness
6         if l − 1 = 0:
7           return false
8         Q.enqueue(s', l−1, d+1)
9       else:
10        F^(l) := F^(l) ∧ ¬s
11        // OPTIONAL: push blocked cube to later frames
12        for i = l...k:
13          if sat(F_0^(i) ∧ T_0,1 ∧ s_1):
14            break
15          else:
16            F^(i) := F^(i) ∧ ¬s
17    return true
```

The function progagate(k) tries to push all blocked cubes from $F^{(k-1)}$ to $F^{(k)}$:

```
1   bool propagate(k):
2     for each blocked cube \lnot s of F^(k-1):
3       if not sat(F_0^(k-1) ∧ T_0,1 ∧ s):
4         F^(k) := F^(k) ∧ ¬s
5     if F^(k-1) = F^(k):
6       return true
7     return false
```

2.3.5 Correctness and Completeness of IC3

In order to be correct, IC3 needs to produce correct counterexamples if a bad state is reachable and correct 1-inductive invariants if that is not the case. This section establishes some properties for the frames and uses them to prove the correctness and completeness of the approach.

Theorem 2.3.16. If IC3 returns false, there is a path from the initial state to a bad state.

Proof. ic3() returns false if base_cases() returns false. That is the case if either the BMC_0 or the BMC_1 formulae is satisfiable, meaning that a counterexample of length 0 or 1 is found. Furthermore, ic3() returns false if strengthen() returns false. The latter one returns false only if a proof obligation has a predecessor in $F^{(0)}$. On the other hand, a proof obligation always leads to a bad state. Because $F^{(0)}$ contains only the initial state, a bad state is reachable from it. □

In the following, it is assumed that frame $F^{(N)}$ is the last present frame.

Theorem 2.3.17. The initial state is contained in every frame, i.e. $I \rightarrow F^{(i)}$ for $0 \leq i \leq N$.

Proof. $F^{(0)} = I$ holds by construction.

Assume that the initial state is not part of frame $F^{(i)}$. Before IC3 is executed, the BMC_0 formula is solved. If it is satisfiable, the execution is aborted and false is returned. Thus, if IC3 creates the first frames, BMC_0 must be unsatisfiable, meaning that the initial state is good. The initial state must be part of the frame when it is created, because the frame is initialized with P. Assuming it is later not part of the frame anymore, it must be removed by a blocked cube. This cannot happen in frame $F^{(0)}$, because this frame has no predecessor frame and therefore cannot have blocked cubes. This means that the number of the frame that excludes the initial state is at least 1. A blocked cube, however, is only added if the state itself is an obligation. This means that a path from that initial to a bad state has been found. This path, however, is shorter than the number of present frames and must therefore be found with a smaller number of frames, meaning that a frame has been created before all transitions from the previous frame to the bad states have been investigated. This is a contradiction to the IC3 algorithm. □

Theorem 2.3.18. All states of each frame are good, i.e., $F_0^{(i)} \rightarrow P_0$ for $0 \leq i \leq N$

Proof. If frames are created, the BMC_0 formula must be unsatisfiable. This means that the initial state (the only state in $F^{(0)}$) is good; therefore, $F^{(0)}$ only contains good states. Every other frame is initialized with P. After that, states are removed but not added. This implies each frame contains only good states. □

Theorem 2.3.19. The sequence of frames grows monotonically, i.e. $F_0^{(i)} \rightarrow F_1^{(i+1)}$ for $0 \leq i < N$.

Proof. $F^{(0)}$ only contains the initial state, and Theorem 2.3.17 shows that this state is also contained in $F^{(1)}$.

Assume that the proposition holds for each frame $F^{(j)}$ with $j < i$ and that there is a state s which is part of $F^{(i)}$ but not of $F^{(i+1)}$. Because of Theorem 2.3.18, s must be a good state.

This means that s must be a proof obligation that is unreachable from $F^{(i)}$ (and hence turned into a blocked cube for $F^{(i+1)}$). In this case, however, s must also be a proof obligation from $F^{(i)}$, because the path leading from s to the bad state can also be started in the latter frame. Because the frame $F^{(i+1)}$ is opened after all proof obligations at $F^{(i)}$ have been investigated, this proof obligation must either be reachable from $F^{(0)}$ (in which case a counterexample has been found and the execution is terminated) or be excluded using a blocked cube in $F^{(i)}$ (which violates the assumption that s is a state of $F^{(i)}$). □

Theorem 2.3.20. Each frame $F^{(i)}$ is an over-approximation of the states reachable after at most i steps, i.e. $F_0^{(i-1)} \wedge T_{0,1} \rightarrow F_1^{(i)}$.

Proof. Assume that there is a state s that is reachable after i steps, but is not part of the frame $F^{(i)}$. Furthermore, assume that i is the smallest frame with this property, i.e. the proposition holds for all $F^{(j)}$ with $j < i$. Theorem 2.3.19 implies that $F^{(i)}$ must contain at least all states from $F^{(j)}$. Because the proposition holds for this frame and its ancestors, $F^{(i+1)}$ contains at least all states reachable after i steps. Furthermore, $F^{(i)}$ is i steps away from $F^{(0)}$. If there is a path of length i from the initial state to any good state s, s cannot be excluded from $F^{(i)}$ (because the proof obligation from which the blocked cube is created is reachable from the initial state). This implies that all good states reachable after i steps must also be part of $F^{(i)}$. If s is a bad state reachable after i steps, the path from the initial state to s can be found at least in frame $F^{(i-1)}$. In this case, the execution is aborted before opening $F^{(i)}$. This implies that $F^{(i)}$ contains any state reachable after at most i steps. □

Theorem 2.3.21. If IC3 returns true, no bad states are reachable.

Proof. ic3 () only returns true if all blocked cubes are pushed from a frame $F^{(i-1)}$ to its successor frame $F^{(i)}$. In this case, both frames contain the same states. With Theorem 2.3.20, it can be concluded that the set of states reachable after $i-1$ steps is equal to the set of states reachable after i steps. Because the transition relation is deterministic, i.e. it can perform the same transitions in each frame, it follows that the frames describe all reachable states in the given SUT. Furthermore, the frames are restricted to the set of good states. Both statements together imply that all reachable states in the SUT are good states. □

It is now clear that IC3 is a correct and complete algorithm for finding counterexamples: If there is a counterexample, IC3 finds it, and any counterexamples given by IC3 are correct (assuming a correct and complete SMT solver). Considering the runtime, the mentioned procedure has a very bad performance. The reason is that each proof obligation and blocked cube only works with a single state. The key idea to make IC3 applicable in practice is to *generalize* them to set of states. The next sections describe how this can be done.

2.3.6 IC3 generalization

The first generalization idea was already proposed by Aron Bradley in his original IC3 paper [11]. Thus, the term IC3 usually incorporates this generalization.

Consider a blocked cube s. Because SMT formulae are used, this blocked cube is described by a set of equality constraints over the state variables.

Example 2.3.22. A blocked cube over a SUT with the integer variables x, y, z may consist of the constraints $\{x = 1, y = 23, z = 42\}$.

The idea of the generalization is to *relax* this blocked cube by removing some of the constraints. This yields a symbolic representation of a *set of states*.

Example 2.3.23. By removing the constraint $x = 1$ from the set of constraints in Example 2.3.22, a set of states is obtained containing each state with $y = 23$ and $z = 42$. All states fulfilling these constraints are contained in the set of states, regardless of the value of x.

This relaxation is done in a way that preserves the property of the blocked cube: each state must be unreachable from the previous frame.

Example 2.3.24. Assume that the obligation from Example 2.3.22 is a predecessor of the obligation o. The constraint $x = 1$ can only be removed if regardless of the value of x, all states fulfilling the other constraints are unreachable from the previous frame.

If a blocked cube o is created at level i, the formula $F_0^{(i-1)} \wedge \neg o_0 \wedge T_{0,1} \wedge o_1$ is unsatisfiable. An SMT solver is able to provide an *unsatisfiable core*. It contains a subset of the original constraints that already renders the SMT formula unsatisfiable. This is similar to a *final conflict clause* in SAT solvers. SMT-LIB provides the (check−sat−assuming) instruction that expects a (possibly empty) set of assumptions and computes an unsatisfiable core from these assumptions. Only the constraints in this core are kept; all others are removed.

This generalization procedure - also referred to as *lifting* - only ensures that all states in the blocked cube are unreachable from the previous frame. It may thus happen that this generalized obligation contains an initial state. However, the exclusion of the initial state from the frame would lead to a violation of Theorem 2.3.17 and must therefore be avoided. Therefore, it must be ensured that the initial state described by I is not contained in the generalized blocked cube q, i.e. $I \wedge q_0$ must be unsatisfiable. Throughout this thesis, the formula $I \wedge o_0$ (using the original proof obligation) is used. This formula must be unsatisfiable, because the proof obligation itself cannot be the initial state (cf. Theorem 2.3.17). An unsatisfiable core of the solver reveals a set of constraints from o that are sufficient to exclude the initial state. These constraints are added to the generalized obligation (i.e. the obligation is *ungeneralized*).

More formally, the generalization is incorporated into IC3 by changing the resolve_obligations() function:

```
1  bool resolve_obligations(k, Q):
2    while Q contains obligation with d < k:
3      (s, l, d) := Q.dequeue()
4      if sat(F_0^(l-1) ∧ ¬s_0 ∧ T_0,1 ∧ s_1):
5        s' := the predecessor state extracted from the witness
6        if l − 1 = 0:
7          return false
8        Q.enqueue(s', l−1, d+1)
9      else:
10       t := the set of states extracted from the unsat core
11       assert(not(sat(I ∧ s_0)))
12       u := the set of states extracted from the unsat core
13       t := t ∩ u
14       F^(l) := F^(l) ∧ ¬t
15       for i = l...k:
16         if not sat(F_0^(i) ∧ ¬t_0 ∧ T_0,1 ∧ t_1):
17           F^(i) := F^(i) ∧ ¬t
18   return true
```

It is worth noting that the unsatisfiable core of the SMT solver does not need to be minimal. Common SMT solvers try to get these cores as small as possible, but an optimal solution is not guaranteed. For IC3, however, the quality of the generalization is of utmost importance: With a good generalization, a lot of states may be investigated at once, leading to a much better overall performance. Later sections introduce approaches to further relax the cores given by the SMT solvers.

Example 2.3.25. Assume that the constraint for x is unnecessary to prove the unsatisfiability of the formula in Example 2.3.22. The solver may thus remove the constraint for x to generalize the set of states. It may also return the complete set of constraints, in which case the blocked cube still describes a single state.

2.3.7 Correctness and Completeness of IC3 generalization

Theorem 2.3.16 is unaffected by the IC3 generalization, because the proof obligations are not touched. The same holds for Theorem 2.3.18, because the generalization just leads to more states being excluded at the same time. The ungeneralization ensures that the initial state is not excluded from any frame (Theorem 2.3.17). The frames still grow monotonically (Theorem 2.3.19): Every state in a generalized blocked cube must be a blocked cube by itself, leading to the same contradiction as in the non-generalized approach. Theorem 2.3.20 holds, because all states in the generalized blocked cube are unreachable from the previous frame (and hence also from frame 0).

Finally, Theorem 2.3.21 is fulfilled because the properties of the blocked cube does not change during the generalization. Because all invariants are fulfilled, IC3 with this generalization is a correct and complete algorithm.

2.3.8 Property Directed Reachability (PDR)

The authors of [22] proposed some extensions to improve the performance of IC3 further. One of their ideas is often incorporated into IC3 algorithms; this extended approach is known as *Property Directed Reachability (PDR)*. While the basic IC3 approach just generalizes blocked cubes, PDR also tries to relax proof obligations: Consider a particular proof obligation o. This proof obligation is a predecessor of a bad state or another proof obligation q. The idea is to remove constraints from o while still ensuring that every state in the resulting set still reaches q with one transition.

The PDR generalization is integrated into the resolve_obligations function, just like the IC3 generalization:

```
1  bool resolve_obligations(k, Q):
2    while Q contains obligation with d < k:
3      (s, l, d) := Q.dequeue()
4      if sat(F_0^{l-1} ∧ ¬s_0 ∧ T_{0,1} ∧ s_1):
5        s' := the predecessor state extracted from the witness
6        if l − 1 = 0:
7          return false
8        t := generalize_obligation(s')
9        Q.enqueue(t, l−1, d+1)
10     else:
11       t := the set of states extracted from the unsat core
12       F^l := F^l ∧ ¬t
13       for i = l...k:
14         if not sat(F_0^i ∧ ¬t_0 ∧ T_{0,1} ∧ t_1):
15           t := the set of states extracted from the unsat core
16           F^l := F^l ∧ ¬t
17   return true
```

In contrast to the IC3 generalization described in Section 2.3.6, it is unclear at a first glance how to obtain a generalization of a proof obligation: The obligation originates from a satisfiable SMT formula for which an unsatisfiable core cannot be created. There are several ways to overcome this problem [40]: The original PDR paper [22] uses ternary simulation. In this thesis, techniques involving SMT solvers are used.

It can be shown that PDR is a correct and complete model checking approach: The property in Theorem 2.3.16 is fulfilled because a bad state can be reached from each state in the generalized obligation. Theorem 2.3.17 holds if the checks to avoid exclusion of the initial state are performed.

Theorem 2.3.18 is unaffected by the switch to PDR. The statement from Theorem 2.3.19 holds because the proof of this statement can be applied to each state in the generalized obligation. Finally, the propositions in Theorem 2.3.20 and Theorem 2.3.21 can be transferred to the PDR case without modifications.

The next section introduces techniques to perform the mentioned PDR generalization as well as further optimizations to other parts of the algorithm.

Chapter 3

Extensions of IC3 and PDR

This section covers techniques for PDR generalization, as well as optimizations of IC3 and PDR. All of them are able to improve the performance of IC3 and PDR in certain situations. A comparison of their performance will follow in Chapter 4.

3.1 Modifications of the SMT formulae

In this section, extensions are presented which modify the SMT formulae to be solved. The goal of the modifications is discussed and advantages as well as disadvantages are presented.

3.1.1 PDR generalization

Section 2.3.8 describes the requirements for a generalization of proof obligations. This section introduces appropriate techniques to perform this generalization [41].

As shown in section Section 2.3.2, proof obligations are created when the reachability of the set of bad states $\neg P$ is checked. There are two places in the IC3 algorithm where proof obligations can be created:

- From a model of the formula $F_0^{(k)} \wedge T_{0,1} \wedge \neg P_1$, in which case the obligation is a direct predecessor of a bad state;
- Using a model for $F_0^{(k)} \wedge \neg s_0 \wedge T_{0,1} \wedge s_1$, implying that the proof obligation is a direct predecessor of another obligation s.

Both have in common that a proof obligation o is created which is a predecessor of a set of states q, i.e., $o_0 \wedge T_{0,1} \wedge q_1$ is satisfiable.

In the original paper about PDR [22], ternary simulation was used for the generalization of proof obligations. For this thesis, however, a different approach is used.

As described in Section 2.3.2, unsatisfiable formulae work well for the generalization of blocked cubes, because solvers can provide unsatisfiable cores for them. A simple idea to transfer this to obligations is to negate the successor state q. The resulting formula $o_0 \wedge T_{0,1} \wedge \neg q_1$, however, is not necessarily unsatisfiable. The reason is that the transition also involves *input variables* for the SUT. These variables also have an influence on the successor state: From a fixed starting state, transitions to several states may be possible using different combinations of values for the inputs of the SUT. The rest of this section deals with the construction of unsatisfiable formulae for given states o, q together with a set of input constraints i.

Generalization with a Negated Proof Obligation (GeNPO)

The straightforward approach to get an unsatisfiable formula for generalization is to also constrain the values of each input according to i. The resulting formula $o_0 \wedge i_0 \wedge T_{0,1} \wedge \neg q_1$ is unsatisfiable if the SUT is *right-unique* (i.e., for each state and each combination of input values, there is a single successor state) [13]. This is the case for the SUTs that BTC Embedded*Platform*® can process. The idea is to solve this formula each time a proof obligation is about to be created. After the solver figured out the unsatisfiability, an unsatisfiable core can be requested. Of course, the obligations still only constrain state variables. If the core contains constraints for input variables, they are not added to the obligation. At a first glance, this might seem like a correct approach to generalize proof obligations.

The problem with this formula, however, is that the SUT may contain *dead-ends*, i.e. states that have no successor. This means that for the dead-end state b, the formula $b \wedge i \wedge T_{0,1}$ is unsatisfiable. Such a dead-end cannot occur in generalized blocked cubes, because these originate from a backwards search starting at the set of bad states. In the case of PDR generalization, a constraint excluding b from the obligation may be removed because the formula is still unsatisfiable. This happens if regardless of the value of b, the described states are dead-ends or predecessors of the state q. Adding b to the proof obligation, however, is incorrect because a state satisfying b does not lead to a bad state. When recursively resolving this incorrect obligation, reaching $F^{(0)}$ does not necessarily imply that the property can be violated: It is possible that none of the paths described by the chain of obligations reaches the bad states, because they all lead to dead-ends.

In order to produce correct results in all cases, the transition must not only be right-unique, but also *left-total*: Each combination of states and inputs must have a successor state. Based on that observation, it is clear that the transition relation must be a *function*. This requirement is acceptable for hardware circuits, which is the reason why many publications about IC3 do not cover this topic at all. BTC Embedded*Platform*®, however, may be faced with dead-ends in the SUTs it analyzes.

It is important to understand that dead-ends have no impact on the correctness of blocked cubes: They are created only if a state is unreachable from the previous frame. The IC3 generalization preseves this property. It is thus correct to exclude each (non-initial) state in the blocked cube, even if it is a dead-end, because all these states are unreachable from the initial state. Another

explanation is that PDR generalization looks forward ("is there a path to a bad state?") and therefore must not exclude dead ends. IC3 generalization, however, looks backwards ("is the state reachable from the previous frame?"), which means that dead-ends do not play a role here. Does this mean that this PDR generalization technique can at least prove unreachability in each case? No! In each run, regardless of the result, a spurious counterexample may occur at any time during the IC3 execution. Thus, in the case of a non-left-total transition relation, this is an approach that is correct regarding the produced invariants, but incorrect regarding the counterexamples being found and imcomplete w.r.t. the returned invariants and counterexamples.

Generalization with a Negated State (GeNSt)

When the IC3 recursion formula $F_0^{(l-1)} \wedge \neg s_0 \wedge T_{0,1} \wedge s_1$, is satisfiable, the solver not only provides a state from the frame $F^{(l-1)}$, but also a state from the set of successor states s (being either a bad state or a state from the successor obligation). The idea of *Generalization with a Negated State (GenSt)* is to use this state instead of the whole set of states to generalize the new obligation. This means that the formula $o_0 \wedge i_0 \wedge T_{0,1} \wedge \neg s_1$ needs to be solved. It might be easier for the solver to prove the unsatisfiability of that formula, because it contains more constraints. However, the generalization performance for the new obligation o may be worse than with GeNPO, because each state in the generalized obligation must be a predecessor of the same single state (instead of the complete obligation). Note that although the generalized obligation is not used in PDR generalization anymore, the IC3 generalization is improved by using sets of states. To the best of the author's knowledge, this approach has not been evaluated before.

Like in GeNPO, invalid proof obligations may be produced if the formula is applied to a SUT with a transition relation not being left-total. The reason is that the front portion of the formula stayed the same. The next section covers an approach that can be used for SUTs with more general transition relations.

Generalization with a Negated Transition Relation (GeNTR)

The previous approaches created an unsatisfiable formula by negating the set of constraints for step 1 (describing a single state in GeNSt or a set of states in GeNPO). One might ask here what happens if other parts of the formula are negated instead. Negating the first part $o_0 \wedge i_0$ does not make sense, because it describes the state to generalize. Another possibility is to negate the transition relation T. Solving the resulting formula $o_0 \wedge i_0 \wedge \neg T_{0,1} \wedge q_1$ is known as *Generalization with a Negated Transition Relation (GeNTR)* [37]. What does "negating the transition relation" mean? In finite SUTs, the normal transition relation T can be thought of as a finite directed graph. The edges of this graph are labeled with the input values to perform the transition. The graph for the negated transition relation is obtained by *complementing* the set of edges. This means inserting exactly those edges that are not present in T. It can be seen that this formula is unsatisfiable, because all edges from o to q are not present in the negated transition relation. Furthermore, this formula is satisfied by dead-end states: In the negated transition relation, such a state has

outgoing edges to *each* state, in particular to each state in q. Thus, the generalization does not add dead-end states from the generalized obligation. This means that PDR with GeNTR is a correct and complete approach even on SUTs that are not left-total. The drawback of this approach is that the GeNTR formula is logically weaker than the GeNPO formula because the negated transition relation contains way more possible transitions than the normal transition relation. Thus, the number of removed constraints is often smaller when using GeNTR. It remains the question how to negate the transition relation. The next section considers the encoding of the transition relation in detail.

3.1.2 Encoding of the Transition Relation

As shown in Section 2.1.6, the transition relation is encoded using a set of equality constraints in SMT-LIB. These equality constraints express assignments in the elaborated SMI program that is used to create the SMT-LIB files.

The variables on the left-hand side of these assignments play different roles: There are state and output variables to encode the respective entities of the SUT. Most of the variables in a SMI program for a complex SUT, however, are auxiliary variables: Converting a program into an SMT formula yields very complex (assert) statements. In particular, the if-then-else (ITE) operator is used heavily to encode e.g. branches and loops. To simplify these statements, the SMI toolchain introduces auxiliary variables containing small parts of the expression. Using further auxiliary variables, these parts are combined to finally express the complete constraint encoded by the (assert) statement.

A simple approach to negate the transition relation is to negate the conjunction of state constraints. The constraints for auxiliary variables are not modified, because they shall still represent the same partial expressions. The original unnegated constraints are still needed for other parts of the PDR algorithm, e.g. finding predecessors of bad states. Therefore, two new propositional variables transition_enable and negated_transition_enable are defined. These are prepended to each state constraint: (assert (= ...)) becomes (assert (or (not transition_enable) (= ...))) and (assert (not (= ...))) becomes (assert (or (not negated_transition_enable) (not (= ...))))). If the variable transition_enable is set to true, (not transition_enable) becomes false and due to the or operator, the equality constraint is evaluated. If transition_enable is set to false, (not transition_enable) evaluates to true and the equality constraint is skipped. Similarly, negated_transition_enable enables or disables the negated transition in the SMT formula. Disabling both the transition and the negated transition allows to perform the IC3 ungeneralization (i.e. avoid exclusion of the initial state from the current frame), because for this check, the transition relation is not needed. This approach has the disadvantage that the constraints of all state variables are encoded twice.

To see how this can be avoided, consider the logical XOR operator. This operator only evaluates to true if either its first or its second operand (but not both!) evaluates to true. If an XOR is used in an (assert ...) statement and the first operand is set to true, the second operand is negated,

because an (assert ...) statement must be fulfilled. By using the negated_transition_enable variable as the first and the conjunction of all state variable assertions as the second operand, the negation of the transition relation can be controlled. To disable the transition relation completely if transition_enable as well as negated_transition_enable are set to false, both enable variables (negated and connected by an AND operator) are prepended to the second operand of the XOR using an OR operator. The resulting formula has the following structure:

```
(assert (xor negated_transition_enable
(or (and (not transition_enable) (not negated_transition_enable)) ...))
```

This approach is referred to as "XOR encoding" throughout this thesis.

The next section introduces a further technique to express sets of states.

3.1.3 Encoding Sets of States

In Section 2.3.2, it was shown that IC3 and PDR operate on the states of the SUT: The basic data structures *frame*, *proof obligation*, and *blocked cube* only constrain the values of state variables.

It was also argued that the generalization is a key factor for the performance: Improving the generalization ability means that IC3 can consider larger areas in the state space at once. In Section 2.3.6 and Section 2.3.8, a single constraint is used to restrict the value of a state variable to encode single states. Generalization is done by removing some of these constraints entirely. This means that after generalization, the set of state variables is divided into two groups: The variables in the first group are restricted to a single value; no generalization is possible. The restrictions for variables in the second group are removed completely: These variables can be set to any value, and still, the resulting state is contained in the set of states. Such variables are often called “don’t cares”; one might say that this is the strongest possible generalization of the state constraint. In short: The generalization for each variable can only happen in a boolean manner; a state variable can be completely unrestricted or bound to a single value.

To understand the basic idea to improve the generalization, consider a modern CDCL SAT solver. In short, such a solver performs *deductions* as well as *decisions* on the propositional variables. The goal of Interval Constraint Propagation (ICP) [20, 7] is to lift this to the SMT level. ICP makes decisions by splitting the intervals of theory variables and is able to propagate literals through the SMT formula. The SMT solver iSAT3 [36] used by BTC Embedded*Platform*® incorporates ICP.

Implementing a whole ICP solver into the IC3 / PDR framework is unfeasible for a master’s thesis. Therefore, a simpler approach is tried that incorporates some ideas from ICP: when encoding a fixed value for a state variable x and a value a, two constraints $x \geq a$ and $x \leq a$ are added. Together, they encode the same requirement as the previous single equality constraints – at least for bit-vector state variables. In floating-point constraints, the floating-point *Not a Number (NaN)* value has to be excluded as well. This encoding has the advantage that generalization can remove one constraint while preserving the other. This may improve the results of generalization and make

IC3 and PDR more performant. By removing both constraints, the variable can be turned into a "don't care", as with the equality encoding. The correctness of IC3 and PDR is not affected by this extension. Proof obligations stay correct, because each state in it still leads to a bad state. Similarly, blocked cubes are correct, because all states are still unreachable from the previous frame.

The next section introduces an extension of IC3 that considers longer paths of the SUT.

3.1.4 Target Enlargement

As explained in Section 2.3.2, the basic IC3 / PDR framework considers only one transition relation at any time. This may help to improve the performance in comparison to k-Induction. However, this may be disadvantageous for SUTs and properties requiring more than one step to be solved. Imagine a property that needs frames $F^{(0)}$, $F^{(1)}$, and $F^{(2)}$. Assume that PDR creates a lot of blocked cube on frame 1. After opening frame 2, all these blocked cubes must be pushed. Although the formulae to solve are often easier than general k-Induction formulae, the large number of proof obligations and blocked cubes may have a significant impact on the solving time.

The idea of *Target Enlargement* [26] is to encode additional transition relations into the property P. The generation of the initial proof obligation now behaves differently: Instead of searching a path of length 1 from the last frame to the bad states, it is now tried to find a path of length $k+1$, where k is the number of additional transition relations. Using this technique, results may be gained in smaller frames. However, the formula to solve for the transition from the last frame to the bad states is now a $k+1$-induction instead of a 1-induction formula. Target Enlargement can therefore be seen as a tradeoff between the complexity of the generation of initial proof obligations and the number of obligations / blocked cubes to process.

Because Target Enlargement incorporates $(k+1)$-induction instead of 1-induction, the BMC formula up to $k+1$ must be solved before executing IC3. For example, with $k = 1$, the BMC_2 formula needs to be solved additionally. The reason is that IC3 starts with two frames and can therefore only detect counterexamples of at least length $k+2$. Under this condition, IC3 / PDR with Target Enlargement is correct and complete regarding properties that can be violated. The produced blocked cubes are also valid, because they continue to remove only unreachable states from the frames. This means that IC3 / PDR with Target Enlargement is also a correct and complete approach to prove invariants.

The next section deals with an approach to improve the results of a generalization further.

3.2 Modifications of the solving process

This section introduces variants of the solving process that may either help to improve the performance of the solvers or allow to ensure the correctness of the results.

3.2.1 Enlarging Generalized Sets of States

The approaches described in Section 3.1.2 and Section 3.1.3 aim at a better generalization performance by improving the formula to be solved. This section deals with improving an obtained generalization result further.

As described in Section 2.3.6, an unsatisfiable core provided by the solver is used to perform the generalization. These unsatisfiable cores, however, do not necessarily need to be minimal: It may be possible to reduce their size by removing some of the remaining constraints or by choosing a different unsatisfiable core containing other constraints.

In general, finding an optimal unsatisfiable core is a difficult problem. For the case of a purely propositional SUT, this is a MaxQBF problem, i.e., an optimization problem over Quantified Boolean Formulae, which is PSPACE-hard [40]. To avoid solving a MaxQBF problem, *approximative procedures* can be used. These approaches do not necessarily produce a minimal unsatisfiable core (just like the SMT solver). However, they are able to remove further literals in an acceptable amount of time.

Removing further constraints from an unsatisfiable core is the basic idea for *Literal Dropping* [35]. One of the constraints from the core is selected; this literal is then removed from the core. At first, it is ensured that the remaining core still excludes the initial states. This can be done by solving the formula $I \wedge r$, where r is the remaining core. If the initial state is not contained in the cube, the formula used to perform the first generalization (i.e. the GeNTR, GeNSt or GeNObl formula) is solved again. Instead of the complete state, only the remaining constraints of the blocked cube are used. If this formula is still unsatisfiable, the literal can be removed from the unsatisfiable core. This procedure can be repeated for each constraint of the blocked cube. However, checking each literal for each blocked cube to generalize is often infeasible. Instead, heuristics are applied to abort literal dropping early. Simple ideas include limiting the number of generalization attempts for a single blocked cube or the number of failing attempts in a row.

Literal dropping can be extended to the down algorithm [12]. Combining down with a technique called Counterexample To Generalization (CTG) yields the ctgDown algorithm [27]. Using *Literal Rotation* [37], the constraints are passed to the solver in different orders, which may influence the order of the decisions the underlying SAT solver makes. From these examples, it can be seen that improving the generalization results is a major research topic [40].

The next section introduces an approach to reduce the runtimes of the satisfiability checks by incorporating multiple instances of the SMT solver.

3.2.2 Using Multiple Solver Instances

During execution of IC3 and PDR, different formula types need to be solved. These types include formulae without, with a negated, and with a normal transition relation, as well as the additional transition relations used for Target Enlargement. In Section 3.1.2 and Section 3.1.4, additional variables are introduced to enable or disable the different formula parts. A different approach is

to start four instances of the SMT solver, one for each formula kind. For this technique, enable variables are not needed. Furthermore, the large formula parts containing the transition relation(s) are never disabled, and thus, the SMT solver does not need to remove the knowledge being learned about the transition. Still, IC3 and PDR are executed sequentially; only one of the solver instances is active at a time. The disadvantage of this approach is an increased RAM usage, because four solver instances are started.

The next section introduces ideas for the certification of the IC3 / PDR results using the available SMT solvers.

3.2.3 Certification of Obligations, Cubes and Invariants

In the previous section, it was shown that IC3 and PDR are sound and complete approaches for Model Checking of embedded systems. However, it was assumed that the underlying SMT solver provides correct results. This means that for satisfiable formulae, the solver must provide a valid model. For unsatisfiable formulae, the solver needs to find valid unsatisfiable cores.

An important observation is that even widely-used SMT solvers contain bugs. Because of the large amount of theories they support and the numerous heuristics they incorporate, these solvers are very complex.

To tackle these challenges, standard software testing approaches like *unit tests* and *regression tests* are used. Furthermore, SMT solvers can be tested on system level by providing formulae with a known result (so-called *oracle-guided approach*). In practice, however, the satisfiability of the formula to solve is often unknown. Therefore, SMT solvers are often tested using *soundness checks*.

A simple approach for such checks is to compare the overall results of the IC3 / PDR execution: If one solver detects a violation of the property while the other one claims that the formula cannot be violated, one of the solvers must contain a bug. It is important to notice that this testing approch does not show which result is actually correct. To answer the question which solver might contain a bug, multiple SMT solvers are often used in parallel. If one of the solvers produces a result that contradicts with the outcome of all other solvers, it is likely that the solving process of the deviating solver is spurious. However, it may also happen that each of the other solver contains a bug and the result of the single solver is actually correct. Nevertheless, soundness checks are the most common technique for testing SMT solvers.

IC3 and PDR do not work with single monolithic SMT formulae. Instead, the SMT formulae being solved produce intermediate results. These are incorporated into further SMT formulae. For example, the check for transitions from the last frame to the bad state yields a proof obligation. To determine its reachability, further formulae are solved that incorporate the constraints describing the obligation. In principle, a soundness check could be performed by passing each SMT formula to multiple solvers. However, this approach has some problems: It was already indicated that different SMT solvers may yield different obligations and cubes for the same SUT. Because of that, one solver must be selected which determines the obligations and cubes used during the IC3 / PDR

execution. This solver is called *primary solver*. All other solvers to be used for the soundness checks are called *secondary solvers*. These solvers solve all formulae that occur during IC3 execution. If the results differ regarding the satisfiability, it is again clear that one solver contains a bug. The models and unsatisfiable cores gained by the secondary solvers are not used during the execution of IC3 and PDR.

The drawback of this approach is that each formula used during an IC3 / PDR run is solved four times. This would reduce the performance significantly. Furthermore, the satisfiability of the formulae is checked, but not the correctness of the produced models / unsatisfiable cores. To overcome this problem, consider the essential entities created during an IC3 run: proof obligations, blocked cubes and frames. In Section 2.3.2, the properties of these entities are defined: A proof obligation is a set of states from which a bad state can be reached. A blocked cube must be unreachable from the previous frame and must not contain an initial state. Instead of solving each formula, the secondary solvers just certify for every obligation / blocked cube gained by the primary solver, i.e. they check if these properties are fulfilled. For proof obligations o on level n with successor obligation q, the formula $o_0 \wedge T_{0,1} \wedge q_1$ is solved. For blocked cubes c at frame n, it is checked that $F_0^{n-1} \wedge T_{0,1} \wedge c_1$ as well as $I \wedge c$ are unsatisfiable. It is important to notice that this approach does not certify the soundness of each IC3 / PDR formula. One might say that we switched from a *formula-based* to an *entity-based* soundness checking approach. This implies that the new approach cannot detect each unsoundness: It may, for example, happen that a solver produces a spurious blocked cube during generalization, but "accidentally" fixes the problem during the ungeneralization. However, such an event seems very unlikely.

Executing these checks on available benchmarks can help to establish trust into the correctness of the IC3 / PDR implementation. For fast model checking, however, this approach is still too time-consuming. To improve the procedure further, the certification needs to be more coarse: Instead of considering each individual entity, only the overall results are checked, i.e. the produced counterexample or invariant. Counterexamples can be easily checked using *simulation*. This means that the run described by the counterexample is executed. During the execution, the input values for each step is applied. After the last step, it is checked that the property is actually violated. BTC Embedded*Platform*® allows to perform resimulation on the elaborated SMI program using the smisim utility. Validating invariants is more complex: An invariant is created if the property is shown to be fulfilled, i.e. there is no run of the SUT that leads to a bad state. Because this is a statement about each execution path in the SUT, it is impossible to perform a simulation using a single path. Invariants are gained from frames in IC3 and PDR. During execution, these frames are represented as SMT formulae. In Section 2.3.1, it is shown that the properties of an invariant can also be expressed by an SMT formula. The property that the invariant only contains good states holds by construction, because the frames are initialized with the set of good states P. The formulae expressing that the invariant is closed under the transition relation and that the invariant contains the initial state can be solved by the available SMT solvers. Note that this approach relies on the actual SMT-LIB encoding: If this encoding does not represent the SUT appropriately, the

certification may accept spurious invariants. The correctness of the encoding, however, is a key assumption of common model checking approaches.

The next section deals with experimental evaluation of all ideas introduced up to this point.

Chapter 4

Experimental Evaluation

This section deals with the evaluation of IC3 / PDR and the several extensions proposed in Chapter 3.

4.1 General Information

Before presenting the actual results, the benchmark set and the IC3 / PDR implementation, as well as the solvers and the evaluation setup is described.

4.1.1 Benchmark Set and Implementation

To evaluate the performance of the IC3 / PDR implementation, the benchmark set from [31] is used. These benchmarks originate from eleven Simulink® models developed for the automotive domain. Using TargetLink®, they are translated to C code. This C code forms an SUT as defined in Section 2.1.1 and can be processed by BTC Embedded*Platform*®. One important requirement for testing these systems is a large *code coverage*. BTC Embedded*Platform*® allows to automatically create *coverage goals*, i.e. properties stating that "the condition in line 23 never evaluates to true" or "the statement in line 42 cannot be reached". Violating such a property means finding a run of the program in which the respective code fragment is reached. If the property always holds, this fragment is shown to be unreachable.

In section Section 2.1.6 it was shown that C programs can be automatically translated to *elaborated SMI* instances. For the SUTs mentioned above, these elaborated SMI encodings are available. To translate the elaborated SMI to SMT-LIB, a program called *smi2engine* is invoked before starting the actual solving.

Table 4.1 presents the number of benchmarks with a particular number of integer / floating-point state variables. In addition, the benchmarks contain up to 1600 boolean state variables. It can be seen that the number of state variables (and, consequently, the size of the state space) is quite large for some of the benchmarks. Furthermore, note that the given numbers hold for a

Integer state variables	Floating-point state variables											
	0	1	2-4	5-9	10-14	15-19	20-34	35-49	50-74	75-99	100-149	$\sum$
0	368	127	218	123	22	14	0	0	0	0	0	872
1	29	0	29	66	128	16	8	0	0	0	0	276
2-4	503	65	117	103	49	91	8	0	0	0	0	936
5-9	242	3	371	555	0	0	2	0	0	0	0	1173
10-14	64	0	32	0	0	0	0	0	0	0	0	96
15-19	21	0	20	0	0	0	0	0	0	0	0	41
20-34	2	0	46	0	0	0	0	0	0	0	78	126
35-49	0	0	27	29	0	775	452	0	0	0	0	1283
50-74	0	0	24	0	0	0	2392	452	0	0	0	2868
75-99	0	0	0	0	0	0	0	0	0	0	0	0
100-149	0	0	0	983	0	0	0	0	0	0	0	983
140-199	0	0	0	0	0	0	0	0	0	0	0	0
200-249	0	0	0	0	0	0	0	0	0	0	0	0
250+	0	0	39	0	38	39	8	0	0	0	0	124
$\sum$	1229	195	923	1859	237	935	2870	452	0	0	78	8778

Table 4.1: Number of integer / floating-point state variables in each benchmark.

single step. If BMC is performed with, e.g., 50 steps, the given numbers must be multiplied by 50 in order to obtain the total number of state variables.

The IC3 / PDR implementation itself is written in Java. The reason is that IC3 requires advanced data structures like HashMaps or PriorityQueues that are not easily available in C. Using Java has the disadvantage that the IC3 implementation cannot use a tailored memory management. It can be expected that the results can be improved at least slightly by implementing the same ideas in C, C++ or even Rust. The Java program starts the solver binary and connects to its standard input and standard output using pipes. This way, the Java program can provide the input and read the solver output.

4.1.2 Solvers and Evaluation Setup

The evaluation is performed on SMT solvers that support SMT-LIB and perform exact reasoning for the theory QF_BVFP, i.e. the logic of quantifier-free bit-vectors and floating-points. Thus, SMT solvers that approximate floating-points with real numbers are not considered here. Furthermore, only complete solvers are investigated, i.e. solvers that are able to show the satisfiability as well as the unsatisfiability of the given formula. As a consequence, the solvers XSAT [25] and GoSAT [6] are not evaluated.

Based on these criteria, the following SMT-LIB compliant solvers are used:

- Bitwuzla [34] in version 0.3.0.

Command: bitwuzla

- CVC5 [2] in version 1.1.2.
  Command: cvc5 −−incremental −−lang=smt2.6 −−output−lang=smt2.6
- MathSAT5 [14] in version 5.6.10.
  Command: mathsat −printer.bv_number_format=2
- Z3 [33] in version 4.12.4.
  Command: z3 model.user_functions=false −−in

The experiments were conducted on a machine with an AMD EPYC® 7542 32-Core CPU running at 2.9 GHz. This CPU launched a Windows® 10 Virtual Machine with 32 virtual cores and 508 GB RAM, executing 32 IC3 / PDR runs in parallel. Windows was used because the utility programs like smi2engine are currently only available for Windows. A timeout of 60s was applied for all experiments unless otherwise stated; the amount of RAM for each solver process was not limited. The number of solved instances per experiment is evaluated. This number is important for Model Checking because the goal is to solve as many instances as possible in a given (short) amount of time. The actual time needed for solving each instance is not quite as important.

If the solver returns a valid counterexample for a given property, this is called a *FALSE* result. A *TRUE* result is produced if the solver returns an invariant. A solver not finishing on a particular instance produces a *TIMEOUT* result. Due to the available amount of RAM, there were no *MEMOUT* results – i.e. no solver aborted its execution due to failed memory allocations. During the experiments, each counterexample found by the solvers was validated by performing a simulation on the SMI representation. The invariants are checked as described in Section 3.2.3. All counterexamples and invariants produced by the solvers are accepted by these checks.

4.1.3 Previous Results

BTC Embedded*Platform*® currently supports the solvers CBMC [38] and iSAT3 [36]. While the former incorporates BMC and k-Induction, the latter uses Craig Interpolation to perform model checking. Furthermore, a prototypical support for the SMT-LIB solvers mentioned above is available, incorporating BMC and k-Induction. The performance of these solvers is evaluted using a timeout of 60 seconds. Table 4.2 presents the results. Bitwuzla solves the largest number of benchmarks, followed by MathSAT5. These two solvers have a large advantage over Z3 and CVC5. It can be seen that the SMT-LIB solvers produce results for the majority of the instances. However, a significant amount of instances cannot be solved even by the fastest solver Bitwuzla. CBMC and iSAT3 perform much better: Both solve 350 more *FALSE* instances than Bitwuzla. iSAT3 is able to provide almost 1000 *TRUE* results. The number of these instances is of utmost importance: Among BMC, BTC Embedded*Platform*® incorporates several techniques to show the violation of a property. For proving the validity of a property, i.e. the unreachability of a code fragment, BTC Embedded*Platform*® solely relies on SMT solvers. If the available solvers are not

able to return results, the unreachability must be investigated manually, which is a difficult task for complex SUTs. Thus, finding new techniques to increase the number of solved *TRUE* instances is very important. Throughout the rest of this section, it is investigated if IC3 and PDR can help to improve the number of solved instances.

Solver	FALSE	TRUE	TIMEOUT
Bitwuzla	7083	474	1221
CVC5	4666	328	3784
MathSAT5	5582	361	2792
Z3	3606	245	4927
CBMC	7419	631	728
iSAT3	7475	972	331

Table 4.2: Previous results on the benchmark set.

4.2 IC3

The evaluation is started with the pure IC3 algorithm (i.e. with generalization of blocked cubes but without generalization of proof obligations). At first, it is checked whether the XOR encoding (cf. Section 3.1.2) improves the performance in comparison to using an explicit negated transition relation. The goal of this experiment is to gain knowledge about the basic IC3 performance. Later, further techniques are added incrementally, allowing to isolate the performance impact of each modification.

The results are presented in Table 4.3. The number of solved instances decreases dramatically in comparison to BMC and k-Induction, especially for the faster SMT solvers Bitwuzla and MathSAT5. Interestingly, MathSAT5 now solves more instances than Bitwuzla and takes the lead. The results show that unoptimized IC3 is not able to solve state-of-the-art model checking problems. Furthermore, it can be seen that the actual encoding of the transition relation has very little impact on the results. Although the negated transition relation is not used for IC3, the XOR encoding will be used for all further experiments, because it does not incorporate two separate encodings of the transition relation.

Section 3.1.3 describes a technique to encode states using $\geq$ / $\leq$ instead of $=$ constraints. Table 4.4 presents the results of the IC3 execution together with this optimization.

The number of benchmarks CVC5 and MathSAT5 solve within the time limit increases significantly compared to the $=$ encoding (+265 resp. +668). Bitwuzla is only able to improve slightly (+36); the distance between this solver and MathSAT5 increases. Furthermore, it can be seen that Z3 solves more *FALSE* instances (+152) but the number of *TRUE* instances decreases (-42). Because the deviation between the different state encodings is noticable, both encodings will be examined during the rest of this thesis.

The overall number of solved instances is the key factor for evaluation of Model Checking

	Bitwuzla			CVC5			MathSAT5			Z3		
	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
BMC / IND	7083	474	1221	4666	328	3784	5582	361	2792	3606	245	4927
separate	3747	298	4733	3129	190	5459	4258	323	4197	2479	221	6078
XOR	3743	301	4734	3111	179	5488	4274	327	4177	2486	218	6074

Table 4.3: Impact of using IC3 in comparison to Bounded Model Checking / k-Induction (BMC / IND), depending of the transition relation encoding.

	Bitwuzla			CVC5			MathSAT5			Z3		
	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
$=$	3743	301	4734	3111	179	5488	4274	327	4177	2486	218	6074
$\geq$ / $\leq$	3777	303	4698	3351	204	5223	4905	364	3509	2638	176	5964

Table 4.4: Impact of the state constraint encoding.

approaches, because the goal is to decide as many problems as possible. However, it gives no insights *why* one approach performs better than another. In Section 2.3.2, it was argued that the IC3 performance is heavily influenced by the generalization capabilities: A slower solver that performs better generalizations may outperform a faster solver, because the former considers less proof obligations and excludes larger blocked cubes. Thus, it seems reasonable to compare the several generalization approaches regarding the average size of the generalized blocked cube / obligation. This means that the number of remaining boolean / bit-vector / floating-point constraints is determined. This number is then divided by the overall number of boolean / bit-vector / floating-point constraints. The resulting numbers are then averaged over the whole experiment. It is also possible to count the number of removed / remaining constraints for the whole experiment and then calculate an average from these numbers. However, this would give more weight to benchmarks with a large number of state variables (and, hence, the possibility to remove many state constraints) or benchmarks with a large number of generalizations. Furthermore, the total number of blocked cubes created by each solver is evaluated. Table 4.5 presents the results.

It can be seen that the generalization capabilities heavily depend on the type of the constraints, as well as on the encoding. All solvers keep more boolean constraints when using the $\geq$ / $\leq$ encoding. Regarding bit-vectors, the amount of needed constraints increases for Z3, but decreases

		Blocked cube				Proof obligation			
		Bool	BV	FP	count	Bool	BV	FP	count
=	Bitwuzla	4.7%	17.7%	6.8%	526170	–	–	–	–
	CVC5	5.3%	27.0%	5.2%	214095	–	–	–	–
	MathSAT5	13.7%	15.3%	7.3%	975511	–	–	–	–
	Z3	13.5%	14.7%	15.8%	508106	–	–	–	–
$\geq$ / $\leq$	Bitwuzla	5.3%	6.1%	3.9%	282421	–	–	–	–
	CVC5	8.8%	8.2%	3.0%	114533	–	–	–	–
	MathSAT5	18.6%	8.1%	5.4%	621793	–	–	–	–
	Z3	17.1%	18.6%	5.1%	337478	–	–	–	–

Table 4.5: Average amount of constraints kept during generalization.

significantly for all other solvers. Similarly, the number of floating-point constraints is reduced with the $\geq$ / $\leq$ encoding. Furthermore, it can be seen that the solver performance does not directly correspond to the generalization performance: the best-performing solver MathSAT5 keeps more boolean variables than other solvers. At a first glance, this might be surprising, because MathSAT5 is the best-performing solver in the presented configurations. However, it needs to be taken into account that the number of remaining constraints in an obligation or a cube is not the only relevant variable: The solver speed and the time needed for generalization also have an impact on the number of solved benchmarks. The results seem to indicate that MathSAT5 is either optimized for the IC3 and PDR formulae types, the generalization approaches of MathSAT5 are very fast, or the solver uses advanced heuristics, helping it to select interesting parts of the search space. This would mean that such solvers may be better suitable to IC3 than a slower solver aiming at a perfect generalization.

In the next section, the performance of PDR is evaluated.

4.3 PDR

This section starts with the evaluation of the several PDR generalization strategies introduced in Section 3.1.1. These strategies are compared on both available encodings of the state constraints. Table 4.6 presents the results.

It can be seen that the solvers are able to solve more *FALSE* instances using PDR, regardless of the generalization approach. The number of solved *TRUE* instances decreases in some configurations, but only slightly. This explains why many publications about IC3 and PDR ignore the IC3 approach entirely and enable both generalization techniques by default. Furthermore, the table illustrates that Generalization with a Negated Proof Obligation (GeNPO) is the best-performing generalization approach, regardless of the state constraint encoding. With this generalization technique, MathSAT5 solves the largest number of benchmarks, followed by Bitwuzla. The gap between them and the two other solvers is very large, especially regarding the number of solved TRUE in-

	Bitwuzla			CVC5			MathSAT5			Z3		
	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
	IC3											
=	3743	301	4734	3111	179	5488	4274	327	4177	2486	218	6074
$\geq$ / $\leq$	3777	303	4698	3351	204	5223	4905	364	3509	2638	176	5964
	Generalization with a Negated Transition Relation (GeNTR)											
=	4219	288	4271	3559	198	5021	5014	498	3266	2899	175	5704
$\geq$ / $\leq$	4885	308	3585	3468	202	5108	5692	502	2584	2923	168	5687
	Generalization with a Negated State (GeNSt)											
=	4088	288	4402	3575	202	5001	4999	473	3306	2922	225	5631
$\geq$ / $\leq$	4818	331	3629	3497	209	5072	5731	522	2525	2937	177	5664
	Generalization with a Negated Proof Obligation (GeNPO)											
=	5215	674	2889	4180	363	4235	5620	696	2462	3417	293	5068
$\geq$ / $\leq$	5872	709	2197	4105	322	4351	6594	794	1420	3840	238	4700

Table 4.6: Impact of the PDR generalization strategies and the state constraint encoding.

stances. Using the other generalization approaches, the gap is smaller. Z3 solves more TRUE instances with equality encoding than with $\geq$ / $\leq$ encoding in all generalization approaches. The results seem to support the assumption from Section 3.1.1: The GeNTR and GeNSt formulae are logically weaker than the GeNPO formula, reducing the generalization capabilities of the solver. To confirm this assumption, the generalization performance is shown in Table 4.7.

It can be seen that Bitwuzla and CVC5 perform the most effective generalizations. Using the equality encoding, CVC5 keeps significantly more constraints than Bitwuzla. As before, the generalizations of MathSAT5 are worse than those of other solvers. Furthermore, the results clearly show the impact of the several PDR generalization strategies: Even GeNTR is able to reduce the size of the obligations significantly. But when using GeNPO combined with the $\geq$ / $\leq$ encoding, the amount of kept constraints is greatly decreased. This corresponds with the solver performance, which is best for the GeNPO approach.

The next section deals with a preprocessing that strengthens the transition relation formula before executing IC3 / PDR.

4.4 Elimination of Constants

The SMT-LIB transition relation may contain *hidden constants*, i.e. state variables with a value that cannot change regardless of the inputs applied to the SUT. The problem with these constants is that the initial state is not present in the k-Induction formula. Thus, the value in step 0 is unknown and hence, it cannot be guaranteed that a particular state variable is indeed a constant.

		Blocked cube				Proof obligation			
		Bool	BV	FP	count	Bool	BV	FP	count
		IC3							
=	Bitwuzla	4.7%	17.7%	6.8%	526170	–	–	–	–
	CVC5	5.3%	27.0%	5.2%	214095	–	–	–	–
	MathSAT5	13.7%	15.3%	7.3%	975511	–	–	–	–
	Z3	13.5%	14.7%	15.8%	508106	–	–	–	–
$\geq$ / $\leq$	Bitwuzla	5.3%	6.1%	3.9%	282421	–	–	–	–
	CVC5	8.8%	8.2%	3.0%	114533	–	–	–	–
	MathSAT5	18.6%	8.1%	5.4%	621793	–	–	–	–
	Z3	17.1%	18.6%	5.1%	337478	–	–	–	–
		Generalization with a Negated Transition Relation (GeNTR)							
=	Bitwuzla	4.0%	19.4%	3.9%	371911	36.5%	67.6%	32.4%	374573
	CVC5	3.8%	35.8%	4.0%	132939	28.5%	80.3%	22.4%	134499
	MathSAT5	9.6%	19.3%	5.5%	653817	72.3%	56.7%	28.0%	657318
	Z3	13.8%	24.1%	9.4%	234905	78.5%	69.2%	35.9%	236125
$\geq$ / $\leq$	Bitwuzla	5.7%	8.5%	2.4%	181144	40.1%	46.3%	33.4%	184470
	CVC5	7.4%	13.5%	3.1%	38356	35.8%	53.1%	24.9%	39821
	MathSAT5	12.9%	10.8%	4.1%	299231	76.7%	41.6%	29.3%	303806
	Z3	13.8%	25.3%	3.4%	126710	79.2%	57.8%	23.8%	127926
		Generalization with a Negated State (GeNSt)							
=	Bitwuzla	4.0%	18.5%	4.3%	348892	36.3%	65.5%	33.1%	351372
	CVC5	4.1%	34.7%	4.8%	126724	29.1%	79.4%	22.7%	128307
	MathSAT5	9.7%	18.7%	5.4%	667645	72.4%	55.6%	27.9%	671064
	Z3	14.2%	24.6%	9.7%	228738	78.4%	68.5%	34.8%	229986
$\geq$ / $\leq$	Bitwuzla	5.5%	7.4%	2.0%	193016	39.7%	42.0%	31.4%	196217
	CVC5	7.4%	13.9%	2.5%	40982	36.1%	53.7%	24.8%	42407
	MathSAT5	13.0%	9.7%	4.5%	316402	77.3%	39.6%	27.7%	320985
	Z3	13.9%	27.2%	2.6%	132784	79.9%	59.5%	23.5%	134072
		Generalization with a Negated Proof Obligation (GeNPO)							
=	Bitwuzla	3.0%	23.6%	2.8%	283878	12.7%	36.5%	6.8%	287977
	CVC5	3.4%	35.3%	2.2%	136461	13.2%	48.3%	4.5%	138900
	MathSAT5	6.1%	21.5%	4.0%	499264	30.2%	32.7%	9.8%	503615
	Z3	9.0%	30.5%	10.4%	242700	34.0%	38.2%	27.5%	244645
$\geq$ / $\leq$	Bitwuzla	5.3%	9.1%	3.5%	115938	18.6%	18.2%	9.8%	121300
	CVC5	5.7%	17.0%	1.9%	54676	13.3%	31.9%	6.4%	57224
	MathSAT5	11.6%	10.9%	3.6%	185001	36.3%	21.3%	10.9%	191207
	Z3	8.6%	31.6%	2.1%	116625	32.3%	35.0%	6.3%	119102

Table 4.7: Average amount of constraints kept during generalization.

The author's bachelor thesis [30] deals with approaches to detect such constants. On the one hand, it describes a syntactic approach for detection of constants: If a variable is initialized with a particular value v during the initialization and in the transition relation, it must be a constant. Furthermore, semantic approaches are presented. The easiest one is called *single elimination*: An

SMT solver checks for each state variable x if the formula $x_0 = v \wedge T_{0,1} \wedge \neg x_1 = v$ is unsatisfiable. If so, the state variable x is shown to be a constant. This procedure is repeated until no further constants are found during a complete iteration. Semantic elimination is able to detect more constants than syntactic elimination. On the other hand, the time needed for syntactic elimination is negligable, whereas semantic elimination needs a significant amount of time. The constants being found are *inlined*, i.e. the definition of the variable is removed and each usage is replaced by the actual value. Using Bitwuzla, semantic elimination can be finished on all benchmarks. This elimination is performed in advance and does not contribute to the execution runtimes. The goal of this approach is to evaluate which results can be gained using a perfect elimination approach.

Table 4.8 presents the results of semantic elimination for the PDR generalization approaches and the available encoding of the state constraints.

		Bitwuzla			CVC5			MathSAT5			Z3		
		FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
		IC3											
no	=	3743	301	4734	3111	179	5488	4274	327	4177	2486	218	6074
	$\geq$ / $\leq$	3777	303	4698	3351	204	5223	4905	364	3509	2638	176	5964
		Generalization with a Negated Transition Relation (GeNTR)											
no	=	4219	288	4271	3559	198	5021	5014	498	3266	2899	175	5704
	$\geq$ / $\leq$	4885	308	3585	3468	202	5108	5692	502	2584	2923	168	5687
syn	=	4119	375	4284	3377	240	5161	4961	550	3267	2937	322	5519
	$\geq$ / $\leq$	5033	468	3277	3491	239	5048	5781	606	2391	2951	263	5564
sem	=	4247	737	3794	3553	454	4771	5069	694	3015	2965	570	5243
	$\geq$ / $\leq$	5068	765	2945	3504	457	4817	5862	748	2168	3098	566	5114
		Generalization with a Negated State (GeNSt)											
no	=	4088	288	4402	3575	202	5001	4999	473	3306	2922	225	5631
	$\geq$ / $\leq$	4818	331	3629	3497	209	5072	5731	522	2525	2937	177	5664
syn	=	3945	347	4486	3188	187	5403	4764	464	3568	2843	290	5645
	$\geq$ / $\leq$	4964	479	3335	3501	239	5038	5907	655	2216	3130	308	5340
sem	=	4126	735	3917	3611	454	4713	5057	719	3002	2923	575	5280
	$\geq$ / $\leq$	5041	770	2967	3503	457	4818	5921	780	2077	3102	594	5082
		Generalization with a Negated Proof Obligation (GeNPO)											
no	=	5215	674	2889	4180	363	4235	5620	696	2462	3417	293	5068
	$\geq$ / $\leq$	5872	709	2197	4105	322	4351	6594	794	1420	3840	238	4700
syn	=	5307	778	2693	4216	397	4165	5648	723	2407	3564	422	4792
	$\geq$ / $\leq$	5988	863	1927	3999	407	4372	6636	852	1290	4036	430	4312
sem	=	5313	884	2581	4225	503	4050	5638	809	2331	3525	646	4607
	$\geq$ / $\leq$	6047	918	1813	4011	497	4270	6624	853	1301	4013	651	4114

Table 4.8: Impact of the elimination of constants.

It can be seen that syntactic elimination of constants leads to an increased number of solved instances. The semantic elimination is able to improve the number of solved instances further. Using syntactic / semantic elimination, Bitwuzla sometimes outperforms MathSAT5 w.r.t. the number of solved TRUE instances, regardless of the state encoding. CVC5 and Z3 are also able to solve more instances. However, the semantic elimination needs a significant amount of time. Integrating semantic elimination into the solver may therefore reduce the number of solved benchmarks again. Thus, it seems reasonable to incorporate only the syntactic elimination approach into the further experiments.

Table 4.9 presents the generalization performance of the experiments when using the syntactic elimination:

		Blocked cube				Proof obligation			
		Bool	BV	FP	count	Bool	BV	FP	count
		Generalization with a Negated Transition Relation (GeNTR)							
$=$	Bitwuzla	4.2%	23.2%	8.2%	352664	35.1%	66.6%	28.7%	355118
	CVC5	3.9%	36.4%	3.4%	104333	27.0%	79.9%	15.4%	105838
	MathSAT5	10.1%	21.2%	10.6%	638837	70.6%	53.8%	23.5%	642203
	Z3	15.3%	22.7%	13.4%	310422	77.2%	60.9%	45.7%	311720
$\geq$ / $\leq$	Bitwuzla	5.8%	11.5%	5.9%	206732	38.8%	48.2%	22.7%	210254
	CVC5	7.2%	13.4%	2.5%	42116	35.3%	54.4%	11.8%	43696
	MathSAT5	13.9%	13.9%	9.4%	327941	75.8%	44.1%	23.1%	332605
	Z3	15.9%	25.1%	6.3%	178908	78.4%	56.7%	24.7%	180339
		Generalization with a Negated State (GeNSt)							
$=$	Bitwuzla	4.1%	22.3%	8.0%	240587	35.7%	67.6%	30.1%	242883
	CVC5	4.1%	36.6%	3.4%	51925	25.6%	82.0%	15.6%	53302
	MathSAT5	10.1%	21.5%	10.7%	497672	71.5%	55.4%	23.1%	500883
	Z3	16.3%	24.6%	12.0%	205129	77.0%	60.3%	45.6%	206280
$\geq$ / $\leq$	Bitwuzla	5.6%	10.4%	3.6%	214856	38.4%	45.2%	21.9%	218369
	CVC5	7.4%	13.4%	2.1%	43379	35.7%	57.0%	11.6%	44935
	MathSAT5	13.7%	12.7%	8.5%	349684	76.1%	42.2%	21.9%	354559
	Z3	16.3%	28.0%	3.6%	174958	79.1%	58.0%	23.7%	176495
		Generalization with a Negated Proof Obligation (GeNPO)							
$=$	Bitwuzla	3.0%	24.3%	4.5%	328732	11.8%	37.0%	5.5%	332991
	CVC5	3.4%	36.8%	2.0%	141392	11.5%	49.6%	3.0%	143926
	MathSAT5	6.3%	20.8%	7.7%	555041	27.8%	31.7%	9.8%	559471
	Z3	9.3%	29.6%	12.2%	306276	31.3%	37.4%	26.2%	308545
$\geq$ / $\leq$	Bitwuzla	5.3%	11.3%	2.8%	114572	17.8%	20.5%	4.7%	120094
	CVC5	6.0%	17.9%	1.8%	53260	17.1%	32.6%	3.5%	55686
	MathSAT5	11.7%	13.5%	3.7%	183660	34.5%	24.2%	7.0%	189949
	Z3	8.2%	31.9%	1.3%	136570	27.9%	34.8%	3.6%	139373

Table 4.9: Average amount of constraints kept during generalization with syntactic elimination of constants.

The percentage of remaining constraints and the total number of obligations / blocked cubes

increases with syntactic elimination. The reason might be that syntactic elimination removes state variables. The constraints belonging to these state variables cannot be removed by the generalization approaches.

The next section deals with the evaluation of Target Enlargement.

4.5 Target Enlargement

As described in Section 3.1.4, Target Enlargement means to encode further transition relations into the negated property. This makes the formula stronger and allows it to create more general obligations. However, it also makes the formula harder to solve. PDR was evaluated with one and two additional transition relations. The results are displayed in Table 4.10.

		Bitwuzla			CVC5			MathSAT5			Z3		
		FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
		Generalization with a Negated Transition Relation (GeNTR)											
T0	=	4119	375	4284	3377	240	5161	4961	550	3267	2937	322	5519
	$\geq$ / $\leq$	5033	468	3277	3491	239	5048	5781	606	2391	2951	263	5564
T1	=	6097	717	1964	4933	375	3470	5991	566	2221	4235	525	2221
	$\geq$ / $\leq$	6227	661	1890	4920	335	3523	6291	557	1930	4298	519	3943
T2	=	6944	628	1206	5425	330	3023	6486	489	1803	4607	504	3667
	$\geq$ / $\leq$	7033	644	1101	5309	275	3194	6595	493	1690	4734	504	3540
		Generalization with a Negated State (GeNSt)											
T0	=	3945	347	4486	3188	187	5403	4764	464	3568	2843	290	5645
	$\geq$ / $\leq$	4964	479	3335	3501	239	5038	5907	655	2216	3130	308	5340
T1	=	6053	631	2094	4928	342	3508	5938	527	2313	4288	533	3957
	$\geq$ / $\leq$	6218	659	1901	4939	335	3504	6285	559	1934	4333	516	3929
T2	=	6938	618	1222	5246	255	3277	6445	463	1870	4649	501	3628
	$\geq$ / $\leq$	7031	642	1105	5310	277	3191	6537	477	1764	4571	490	3717
		Generalization with a Negated Proof Obligation (GeNPO)											
T0	=	5307	778	2693	4216	397	4165	5648	723	2407	3564	422	4792
	$\geq$ / $\leq$	5988	863	1927	3999	407	4372	6636	852	1290	4036	430	4312
T1	=	6402	845	1531	5160	430	3188	6172	640	1966	4479	577	3722
	$\geq$ / $\leq$	6585	831	1362	5122	395	3261	6532	651	1595	4581	579	3618
T2	=	6892	665	1221	5311	266	3201	6447	541	1790	4497	524	3757
	$\geq$ / $\leq$	6926	675	1177	5332	271	3175	6583	569	1626	4630	545	3603

Table 4.10: Impact of target enlargement with 0 / 1 / 2 additional transition relations (T0 / T1 / T2) using the syntactic elimination of constants.

It can be seen that using a target enlargement of 1, the number of solved instances can often be improved. An exception is MathSAT5, which solves more TRUE instances without target

enlargement. Using a target enlargement of 2, the number of solved FALSE instances increases, but the number of solved TRUE instances decreases. The reason for the former observation is that target enlargement requires additional BMC steps to be solved. This implies that counterexamples with a small number of steps can now be detected by BMC. Regarding the latter fact, it seems that a Target Enlargement of 2 (resp. 1 for MathSAT5) yields too complex formulae – at least with a timeout of 60s.

The next section deals with the effect of literal dropping.

4.6 Literal Dropping

In Section 3.2.1, a technique called *literal dropping* is introduced to reduce the size of proof obligations and blocked cubes further. For this thesis, literal dropping is executed using the following policy: At most 30 dropping attempts are made. If two consecutive attempts fail, literal dropping is aborted. The results are presented in Table 4.11.

		Bitwuzla			CVC5			MathSAT5			Z3		
		FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
		Generalization with a Negated Transition Relation (GeNTR)											
-	=	4119	375	4284	3377	240	5161	4961	550	3267	2937	322	5519
-	$\geq$ / $\leq$	5033	468	3277	3491	239	5048	5781	606	2391	2951	263	5564
+	=	4051	395	4332	3519	230	5029	4945	561	3272	3169	337	5272
+	$\geq$ / $\leq$	4724	442	3612	3471	200	5107	5639	631	2508	3104	300	5374
		Generalization with a Negated State (GeNSt)											
-	=	3945	347	4486	3188	187	5403	4764	464	3568	2843	290	5645
-	$\geq$ / $\leq$	4964	479	3335	3501	239	5038	5907	655	2216	3130	308	5340
+	=	4040	386	4352	3375	230	5173	4749	589	3440	3215	337	5226
+	$\geq$ / $\leq$	4588	448	3742	3422	203	5153	5561	637	2580	3146	311	5321
		Generalization with a Negated Proof Obligation (GeNPO)											
-	=	5307	778	2693	4216	397	4165	5648	723	2407	3564	422	4792
-	$\geq$ / $\leq$	5988	863	1927	3999	407	4372	6636	852	1290	4036	430	4312
+	=	5100	701	2977	3903	331	4544	5687	696	2395	4009	432	4337
+	$\geq$ / $\leq$	5738	767	2273	3898	330	4550	6519	817	1442	4348	467	3963

Table 4.11: Comparison of experiments without (-) and with (+) literal dropping.

It can be seen that using literal dropping, the number of solved instances can sometimes be improved. For the best generalization strategy GeNPO, however, the number of solved TRUE instances decreases. It seems that Literal Dropping is more helpful for the weak generalization strategies and has more impact on the number of counterexamples being found. Maybe the number

of 30 dropping attempts is already too large for this benchmark set.

The next section contains the results for the execution on multiple solver instances.

4.7 Using multiple solver instances

Section 3.2.2 explained how to distribute the several formula kinds on different SMT solver instances. This idea is evaluated using the $\geq$ / $\leq$ encoding and the syntactic elimination. Table 4.12 contains the results.

	Bitwuzla			CVC5			MathSAT5			Z3		
	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT	FALSE	TRUE	TIMEOUT
single	5988	863	1927	3999	407	4372	6636	852	1290	4036	430	4312
multi	5967	844	1967	4213	422	4143	6626	861	1291	4088	434	4256

Table 4.12: Impact of using multiple solver instances.

Interestingly, the usage of multiple solver instances has almost no effect on the number of solved instances. In some configurations, it even performs worse than the single instance approach. For the slowest solvers, however, the results are improved a bit by using multiple instances. One can also look at this result differently: It seems that using a single solver instance and assumptions to enable / disable parts of the formula has almost no negative impact on the solver performance.

4.8 1h Timeout

All experiments up to this point are performed with 60s timeout. The results show how many instances the solvers can process in a short amount of time. Another question is how the solvers perform on harder instances if a longer execution is allowed. To analyze this, PDR with the GeNPO strategy and the $\geq$ / $\leq$ encoding is evaluated with a timeout of one hour. The results are presented in Table 4.13.

All SMT-LIB solvers are able to solve significantly more instances after one hour. The same is not true for CBMC and iSAT3; the results of these solvers only improve slightly with a timeout of one hour. In fact, Bitwuzla and MathSAT5 outperform these solvers with a timeout of one hour.

Figure 4.1 presents the number of instances solved after a given amount of time.

It can be seen that Bitwuzla and MathSAT5 solve most of the instances within five minutes. MathSAT5 solves a bit more instances in 1 to 15 minutes. With longer execution times, the performance of both solvers is almost equal. CVC5 and Z3 follow with a large gap. With longer execution times, CVC5 solves some more instances than Z3. The results indicate that for complex goals and

Solver	60s FALSE	60s TRUE	60s TIMEOUT	3600s FALSE	3600s TRUE	3600s TIMEOUT
Bitwuzla	5872	709	2197	7538	1013	227
CVC5	4105	322	4351	6240	833	1705
MathSAT5	6594	794	1420	7542	1007	229
Z3	3840	238	4700	6047	734	1997
CBMC	7419	631	728	7606	632	540
iSAT3	7475	972	331	7537	987	254

Table 4.13: Performance with a timeout of one hour, using the GeNPO strategy and the $\geq$ / $\leq$ encoding.

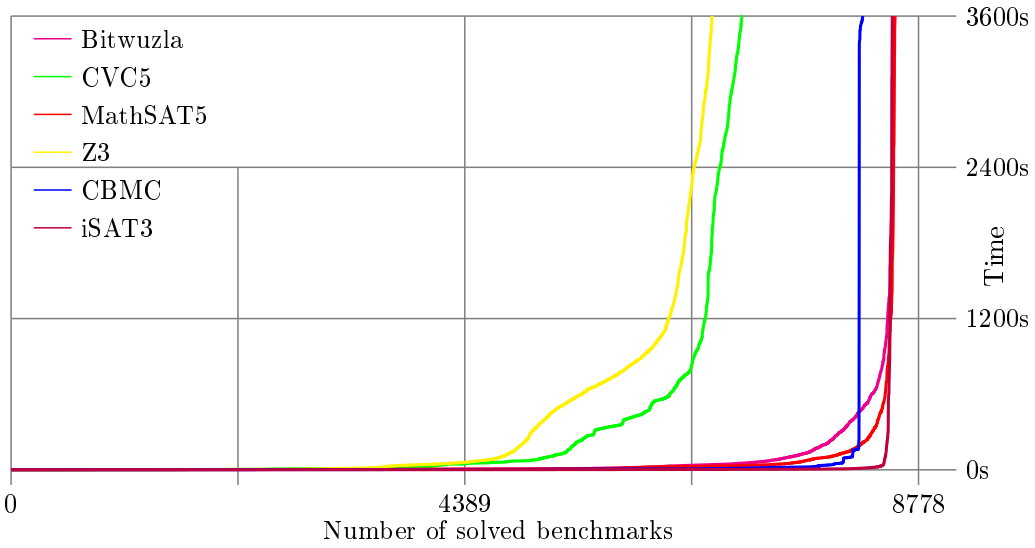


Figure 4.1: Number of solved instances per execution time

longer execution times, PDR executed on SMT-LIB solvers can reach a better performance than the solvers currently used by BTC Embedded*Platform*®. It must be noted, however, that the runtimes depend on the benchmark set. Furthermore, a runtime of 1h may already be too long for some use cases. In this case, the solvers CBMC and iSAT3 are still able to solve more instances than the SMT-LIB solvers.

4.9 Summary

The performance of the SMT-LIB solvers increases significantly when using PDR instead of Bounded Model Checking and k-Induction. Table 4.14 illustrates this by comparing the best results gained by the solvers using the different available solving approaches. The results for BMC / k-Induction originate from [31]. The comparison also includes a PDR implementation in iSAT3

Solver	FALSE	TRUE	TIMEOUT
	BMC / k-Induction		
Bitwuzla	7666	628	484
CVC5	7074	541	1163
MathSAT5	7434	626	718
Z3	6359	577	1842
	PDR		
Bitwuzla	7538	1013	227
CVC5	6240	833	1705
MathSAT5	7542	1007	229
Z3	6047	734	1997
	CBMC / iSAT3		
CBMC	7419	631	728
iSAT3	7475	972	331
iSAT3 + PDR	7622	1003	153

Table 4.14: Best performance gained by the solvers / solving approaches using a timeout of 1h.

[37]. This implementation makes use of iSAT3's builtin support for Interval Constraint Propagation (ICP) and uses Generalization with a Negated Transition Relation (GeNTR). The results are gained with a different measurement setup, but should still show the general trend.

It can be seen that iSAT3 + PDR is able to solve more instances than the standard iSAT3 implementation. However, the number of solved *TRUE* instances is still smaller than the number of such instances solved by Bitwuzla and MathSAT5.

For Model Checking, portfolio approaches are often used, i.e. multiple solvers are executed one by one on the same verification problem until one solver succeeds. Thus, it seems reasonable to compare the instances solved by the SMT-LIB solvers with the combined instances solved by CBMC and iSAT3 – i.e. the two solvers used by BTC Embedded*Platform*®. This comparison shows that the SMT-LIB solvers are not able to gain any "new" results, i.e. they do not solve any instance that is not solved by the portfolio consisting of CBMC and iSAT3. This demonstrates the strength of a portfolio approach: Weaknesses of a single solver in particular areas can be compensated by a different solver.

Chapter 5

Conclusion and Outlook

In this chapter, a summary of the essential contents is provided. Afterwards, ideas are presented that build on the presented approaches.

5.1 Conclusion

This thesis deals with Model Checking of embedded systems written in C. The focus is on detection of *dead code*, i.e. unreachable parts of the source code. This C code can be rewritten to SMT formulae I, T, and P, encoding the initial states, transition relation, and the property to check. This allows to perform Model Checking using an SMT solver. For that purpose, Bounded Model Checking and k-Induction are often used. These approaches perform a breadth-first search over the System Under Test (SUT). Alternatively, IC3 and PDR can be used. The key difference is that these approaches are depth-first search algorithms to traverse the state space of the system. IC3 and PDR work with *frames*, i.e. over-approximations of the set of states reachable after a given number of steps. *Proof obligations* are used to check the reachability of bad states from the frames recursively. If a state is shown to be unreachable from the initial state, it is turned into a *blocked cube*, which means it is excluded from the respective frame. Generalization of proof obligations and blocked cubes is a key factor for the performance of both approaches. In particular, the generalization of obligations (which distinguishes PDR from IC3) imposes challenges. There are several ideas that may help to improve the IC3 / PDR performance further. These ideas include different encodings of the transition relation and the state constraints, Target Enlargement to use more transition relations for the creation of the initial proof obligations, Literal Dropping to improve existing generalizations, the usage of multiple solver instances for different formula kinds, and the certification of intermediate and overall results for checking the reliability of the SMT solvers.

For this thesis, an IC3 / PDR implementation has been developed. It supports both approaches as well as all mentioned variants. All these techniques are evaluated on the SMT-LIB

solvers Bitwuzla, CVC5, MathSAT5, and Z3. The results provided by these solvers are all correct. Literal Dropping and the usage of multiple solver instances have negligable or even negative impact, at least with a timeout of 60 seconds. The $\geq$ / $\leq$ encoding, on the other hand, helps to improve the number of solved instances. The same holds for syntactic elimination and for Target Enlargement with a single transition relation, at least when using Bitwuzla, CVC5 and MathSAT5. The generalization approach Generalization with a Negated Proof Obligation (GeNPO) performs best, with a significant lead over the other approaches Generalization with a Negated State (GeNSt) and Generalization with a Negated Transition Relation (GeNTR). The solvers perform best if Generalization with a Negated Proof Obligation (GeNPO) and the $\geq$ / $\leq$ encoding for state constraints are used together with the syntactic elimination of constants. Using a timeout of one hour, Bitwuzla and MathSAT5 are even able to outperform the solvers CBMC and iSAT3.

5.2 Outlook

The results demonstrate that Model Checking for real-world embedded systems is possible in practice. However, there are Model Checking problems that cannot be solved by any currently available approach. Furthermore, the complexity of the used systems grows continously. In particular, the number of states increases permanently.

This thesis presented the foundations of IC3 and PDR. As already mentioned, improving these algorithms further is an active research topic. There exist many techniques to improve the generalizations provided by the SMT solvers, for example Literal Rotation. These approaches can also be implemented in the IC3 implementation that is used for the experiment in this thesis. In addition, the performance of the Generalization with Negated Proof Obligation (GeNPO) can be evaluated on SUTs that are not left-total (i.e. they contain dead-end states). As stated, the produced invariants remain correct, but spurious counterexamples may occur. The question remains how often such a spurious counterexample is created in practice. It may even be possible to detect incorrectly generalized obligations and "repair" them to ensure they keep the desired properties – i.e. all states in an obligation need to be predecessors of bad states. Furthermore, a more sophisticated Interval Constraint Propagation (ICP) algorithm can be used. The key idea of ICP is to lift the entire reasoning to the theory level. For example, full ICP solvers are able to deduce intervals for arithmetic operations. All ICP techniques have the goal to relax the interval bounds. The good results gained by the $\geq$ / $\leq$ encoding show that such ideas may help the solvers to perform better generalizations. Finally, portfolio approaches incorporating techniques from this thesis can be evaluated. For example, PDR (with the solvers Bitwuzla and / or MathSAT5) could be combined with CBMC and iSAT3. In typical portfolio approaches, each solver gets a specific amount of the overall solving time. If it does not succeed within that time limit, the execution is aborted and the next solver is started. This can be refined e.g. by executing the solver multiple times with increasing timeouts. Such approaches can often gain a better performance than a single solver, because every solver is often tailored to a specific sort of benchmarks. In the long-term,

the heuristics of BTC Embedded*Platform*® can be improved by incorporating approaches and solvers from this thesis. This may help to successfully perform Model Checking for the automotive systems developed in the future.

References

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

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_24.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. URL: www.SMT-LIB.org.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB theories*. URL: www.SMT-LIB.org/theories.shtml.

[5] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825. URL: https://doi.org/10.3233/978-1-58603-929-5-825.

[6] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. "goSAT: Floating-point satisfiability as global optimization". In: *2017 Formal Methods in Computer Aided Design (FMCAD)*. 2017, pp. 11–14. DOI: 10.23919/FMCAD.2017.8102235.

[7] Frédéric Benhamou and Laurent Granvilliers. "Chapter 16 - Continuous and Interval Constraints". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 571–603. DOI: https://doi.org/10.1016/S1574-6526(06)80020-9. URL: https://www.sciencedirect.com/science/article/pii/S1574652606800209.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". In: *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*. Ed. by Mary Jane Irwin. ACM Press, 1999, pp. 317–320. DOI: 10.1145/309847.309942. URL: https://doi.org/10.1145/309847.309942.

[9] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. "CaDiCaL 2.0". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14681. Lecture Notes in Computer Science. Springer, 2024, pp. 133–152. DOI: 10.1007/978-3-031-65627-9_7.

[10] Armin Biere and Mathias Fleury. "Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022". In: *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Vol. B-2022-1. Department of Computer Science Series of Publications B. University of Helsinki, 2022, pp. 10–11.

[11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.

[12] Aaron R. Bradley and Zohar Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Formal Methods in Computer Aided Design (FMCAD'07)*. 2007, pp. 173–180. DOI: 10.1109/FAMCAD.2007.15.

[13] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. "Incremental formal verification of hardware". In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 135–143.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Proceedings of TACAS*. Ed. by Nir Piterman and Scott Smolka. Vol. 7795. LNCS. Springer, 2013.

[15] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. London, Cambridge: MIT Press, 1999. ISBN: 0-262-03270-8.

[16] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods Syst. Des.* 19.1 (2001), pp. 7–34. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260.

[17] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. "Symbolic Model Checking". In: *International Conference on Computer Aided Verification*. 1993. URL: https://api.semanticscholar.org/CorpusID:266032052.

[18] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937. URL: http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937.

[20] Ernest Davis. "Constraint propagation with interval labels". In: *Artificial Intelligence* 32.3 (1987), pp. 281–331. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(87)90091-9. URL: https://www.sciencedirect.com/science/article/pii/0004370287900919.

[21] Martin Davis, George Logemann, and Donald W. Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (1962), pp. 394–397. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[22] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.

[23] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver." In: *SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. ISBN: 3-540-20851-8. URL: http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03.

[24] Niklas Eén and Niklas Sörensson. "Temporal induction by incremental SAT solving". In: *Electron. Notes Theor. Comput. Sci.* 89.4 (2003), pp. 543–560. DOI: 10.1016/S1571-0661(05)82542-3. URL: https://doi.org/10.1016/S1571-0661(05)82542-3.

[25] Zhoulai Fu and Zhendong Su. "XSat: A Fast Floating-Point Satisfiability Solver". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 187–209. ISBN: 978-3-319-41540-6.

[26] Alberto Griggio and Marco Roveri. "Comparing Different Variants of the ic3 Algorithm for Hardware Model Checking". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 35.6 (2016), pp. 1026–1039. DOI: 10.1109/TCAD.2015.2481869. URL: http://dx.doi.org/10.1109/TCAD.2015.2481869.

[27] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. "Better generalization in IC3". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 157–164. DOI: 10.1109/FMCAD.2013.6679405.

[28] ISO. *Road vehicles – Functional safety.* Norm. 2011.

[29] *List of SMT solvers.* URL: www.SMT-LIB.org/solvers.shtml.

[30] Lukas Mentel. "Detection and Elimination of Constants to Strengthen k-Induction". In: (2021), pp. 135–143.

[31] Lukas Mentel, Karsten Scheibler, Felix Winterer, Bernd Becker, and Tino Teige. "Benchmarking SMT Solvers on Automotive Code". In: *MBMV 2021; 24th Workshop.* 2021, pp. 1–10.

[32] E.F. Moore. *The Shortest Path Through a Maze.* Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL: https://books.google.de/books?id=IVZBHAAACAAJ.

[33] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[34] Aina Niemetz and Mathias Preiner. "Bitwuzla". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II.* Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 3–17. DOI: 10.1007/978-3-031-37703-7_1. URL: https://doi.org/10.1007/978-3-031-37703-7%5C_1.

[35] Kavita Ravi and Fabio Somenzi. "Minimal Assignments for Bounded Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Kurt Jensen and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 31–45. ISBN: 978-3-540-24730-2.

[36] Karsten Scheibler. "Applying CDCL to verification and test: when laziness pays off". PhD thesis. University of Freiburg, Freiburg im Breisgau, Germany, 2017. URL: https://freidok.uni-freiburg.de/data/12669.

[37] Karsten Scheibler, Felix Winterer, Tobias Seufert, Tino Teige, Christoph Scholl, and Bernd Becker. "ICP and IC3". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* 2021, pp. 1116–1121. DOI: 10.23919/DATE51398.2021.9473970.

[38] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. *Incremental Bounded Model Checking for Embedded Software (extended version).* 2014. arXiv: 1409.5872 [cs.SE]. URL: https://arxiv.org/abs/1409.5872.

[39] R. Sebastiani. "Lazy Satisfiability Modulo Theories". In: *J. Satisf. Boolean Model. Comput.* 3 (2007), pp. 141–224.

[40] Tobias Seufert. "On safety verification using PDR and Reverse PDR". PhD thesis. University of Freiburg, Freiburg im Breisgau, Germany, 2023. URL: https://freidok.uni-freiburg.de/data/240847.

[41] Tobias Seufert, Felix Winterer, Christoph Scholl, Karsten Scheibler, Tobias Paxian, and Bernd Becker. "Everything You Always Wanted to Know About Generalization of Proof Obligations in PDR". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.4 (2023), pp. 1351–1364. DOI: 10.1109/TCAD.2022.3198260.

[42] João P. Marques Silva and Karem A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability". In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433. URL: https://doi.org/10.1109/12.769433.

[43] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24. URL: https://doi.org/10.1007/978-3-642-02777-2%5C_24.

[44] Cesare Tinelli. *The SMT-LIB Core theory*. URL: www.SMT-LIB.org/theories-core.shtml.

[45] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265. URL: http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

Appendix A

Generated SMT-LIB files

Header File

```
; ---- SMT2 HEADER FILE ----
; target engine           : SMT2
; file name               : example_header.smt2
; original symtab file    : example.symtab
; original smi file       : example.smi
; smi program name        : SUBTASK_1
; purpose                 : bounded model checking
; This file was automatically generated by smi2engine v0.1
; on Sun May 23 18:00:00 2021 (UTC).

; Define the logic (theory) to use
(set-logic QF_BVFP)

; Create model if result is SAT
(set-option :produce-models true)

; Shorthands for operators

; Bool operators

; -- not needed here, omitted --

; BitVec operators
```

```
; -- not needed here, omitted --

; FP to BitVec conversion operators

; -- not needed here, omitted --
```

Declaration File

```
; ---- SMT2 DECLARATION FILE ----
; target engine           : SMT2
; file name               : example_decl.smt2
; original symtab file    : example.symtab
; original smi file       : example.smi
; smi program name        : SUBTASK_1
; purpose                 : bounded model checking
; This file was automatically generated by smi2engine v0.1
; on Sun May 23 18:00:00 2021 (UTC).

; Declaration of variables

; ---- Variable Declaration: counter ----
; Type: BitVec, 32 bits
; Range: [ -2147483648 | 2147483647 ]
(declare-fun counter$$new () (_ BitVec 32))
(assert (bvsge counter$$new #x80000000))
(assert (bvsle counter$$new #x7fffffff))

; ---- Variable Declaration: executions ----
; Type: BitVec, 32 bits
; Range: [ -2147483648 | 2147483647 ]
(declare-fun executions$$new () (_ BitVec 32))
(assert (bvsge executions$$new #x80000000))
(assert (bvsle executions$$new #x7fffffff))

; ---- Variable Declaration: x ----
; Type: BitVec, 32 bits
; Range: [ -2147483648 | 2147483647 ]
(declare-fun x$$new () (_ BitVec 32))
(assert (bvsge x$$new #x80000000))
```

```
(assert (bvsle x$$new #x7fffffff))

; ---- Variable Declaration: invariance_assumption ----
; Type: Bool
(declare-fun invariance_assumption$$new () Bool)

; restricting variables regarding NaN
```

Initialization File

```
; ---- SMT2 INIT FILE ----
; target engine           : SMT2
; file name               : example_init.smt2
; original symtab file    : example.symtab
; original smi file       : example.smi
; smi program name        : SUBTASK_1
; purpose                 : bounded model checking
; This file was automatically generated by smi2engine v0.1
; on Sun May 23 18:00:00 2021 (UTC).

; Initialization of variables

; ---- Variable Definition: counter ----
; Type: BitVec, 32 bits
; Value: 0
(assert (= counter$$0 #x00000000))

; ---- Variable Definition: executions ----
; Type: BitVec, 32 bits
; Value: 0
(assert (= executions$$0 #x00000000))

; ---- Variable Definition: invariance_property ----
; Type: Bool
; Value: true
(assert invariance_property$$0)
```

Transition File

```
; ---- SMT2 TRANSITION FILE ----
; target engine           : SMT2
; file name               : example_transition.smt2
; original symtab file    : example.symtab
; original smi file       : example.smi
; smi program name        : SUBTASK_1
; purpose                 : bounded model checking
; This file was automatically generated by smi2engine v0.1
; on Sun May 23 18:00:00 2021 (UTC).

; Transition Relation

(assert (= counter$$new (ite (distinct x$$new #x00000000)
(bvadd counter$$old x$$new) #x00000000)))

(assert (= executions$$new (bvadd executions$$old #x00000001)))

(assert (= invariance_property$$new (bvslt counter$$new executions$$new))))
```

Property File

```
; ---- SMT2 PROPERTY FILE ----
; target engine           : SMT2
; file name               : example_property.smt2
; original symtab file    : example.symtab
; original smi file       : example.smi
; smi program name        : SUBTASK_1
; purpose                 : bounded model checking
; This file was automatically generated by smi2engine v0.1
; on Sun May 23 18:00:00 2021 (UTC).

; Target Property
(assert invariance_property$$new)
```