Detection and Elimination of Constants to Strengthen k-Induction

UNIKASSEL VERSITÄT

BTC embedded systems

Bachelor Thesis

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor 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 Matriculation Number: 35067548
Examiners:	Prof. Dr. Martin Lange Faculty of Theoretical Computer Science / Formal Methods University of Kassel
	Prof. Dr. Stefan Göller Faculty of Theoretical Computer Science / Complex Systems University of Kassel
Advisor:	Dr. Karsten Scheibler Senior Research Engineer BTC Embedded Systems AG
Submitted:	May 26, 2021

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, May 26, 2021

Lukas Mentel

Contents

1 Introduction

Today, our daily lives are very much influenced by computer systems. More and more of these systems are being given a responsibility for safety: For example, autonomous robots are commanded by software or neural networks. If these systems fail, they may damage other systems or even hurt people. Therefore, the correctness of these systems is very important.

Computer science has studied how to address these risks for a long time. As a result, multiple approaches have been developed. Most of them, like *unit-tests*, work by providing specific inputs to the system and check the results. These approaches suffer from the problem that systems are too complex today to exhaustively test all possible combinations of inputs.

This thesis builds upon a technique called Model Checking [1] that performs a complete search over all possible executions of the system to investigate. In contrast to the testing techniques described above, it allows to prove the validity of *all* possible executions of the system.

This thesis concentrates on *embedded systems*, which are widely used in vehicles, home appliances, industry etc. These systems are not controlled with usual input devices as mouse, keyboard or a touch display, but get data from sensors and other devices such as a brake pedal. From these data and their current internal state, they calculate output values that are provided to, e.g., actors or other computer systems (but not to a screen).

Section 2 introduces model checking, together with the mathematical preliminaries needed for this thesis. In Section 3, a technique is described to identify constants, i.e., variables whose values cannot change, and several strategies are introduced to improve this basic principle further. These strategies are evaluated on a particular benchmark set in Section 4. Section 5 summarizes the results of this thesis and proposes some ideas that could make elimination of constants even more useful.

2 Preliminaries

This section deals with the fundamental concepts that are used within this thesis. It starts with a general introduction to Model Checking and continues with propositional logic, together with the extension Satisfiability Modulo Theories. After introducing the SMT-LIB format, the transformation from (a subclass of) C programs into this format is described. This section ends by presenting Bounded Model Checking and k-Induction.

2.1 Model Checking

This section describes the general idea of Model Checking [1] as an important technique to check a system against a set of given requirements (a *specification*). This specification induces a set of "good" (conforming to the specification) and a set of "bad" (all other) states. Furthermore, a system to be checked consists of an initial state, together with a transition relation that describes how the state of the system changes, dependent on the current inputs. Each sequence of states that starts in the initial state and conforms to the transition relation is called an *execution* of this system.

Model Checking implements an explorative search of the underlying state space. In contrast to techniques such as *fuzzing*, i.e., providing random inputs to a program and see if it handles them correctly [23], all possible execution paths of the system are considered. Because of that, Model Checking is able to prove not only the presence, but also the absence of errors. If the model checker finds an execution of the system that ends in one of the "bad" states, this execution is called a *counterexample* for the property being investigated. If such a counterexample does not exist, it is proven that the given property holds in the underlying system.

The next sections describe logics that are used to (1) model the system to be evaluated, and (2) define the property to be checked.

2.2 Propositional Logic and SAT

This section introduces *propositional logic* [28] as a first step towards a formal representation of systems and their properties in a machine-readable and unambiguous manner.

Example 2.1. This section describes a formalization of statements such as

If a computer is turned on, it consumes power and produces heat.

This statement relates the state of the computer (on / off) to the power consumption (power-consuming, non-power-consuming) and the produced temperature changes.

Note that the statement from Example 2.1 does *not* quantify the amount of power consumption or heat production. All entities (state of computer, power consumption, heat production) can take exactly two different values. A *propositional formula* is built up from such two-value (boolean) entities connected by boolean operators:

Definition 2.2. The set P of propositional formulae over a set of Variables V is defined as follows:

- tt and ff are propositional formulae.
- All variables v in V are propositional formulae.
- For two propositional formulae φ and ψ, $\neg\varphi$ and $\varphi \wedge \psi$ are propositional formulae.

Because a propositional formula may contain variables, it does not make sense to say that such a formula is *true* or *false*. A formula may only be evaluated with if a value is assigned to each variable in that formula. Technically, such a variable assignment is a function $\theta : V \to \{true, false\}$.

The variable assignment can be used to define the interpretation $I : P \to \{true, false\}$ of all possible formulae:

Definition 2.3. For each formula $\varphi, \psi \in P$ and each variable $v \in V$, let

- $I(tt) = true$
- $I(ff) = false$
- $I(v) = \theta(v)$
- $I(\neg\varphi) = true$ iff $I(\varphi) = false$
- $I(\varphi \wedge \psi) = true$ iff $I(\varphi) = true$ and $I(\psi) = true$

There is a set of convenient shorthands for propositional formulae:

Definition 2.4. For all φ, $\psi \in P$ let

- $\varphi \vee \psi \coloneqq \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \to \psi \coloneqq \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \coloneqq \varphi \to \psi \wedge \psi \to \varphi$

Example 2.5. Consider the statement from Example 2.1. It connects three boolean variables C, P, and H, where C represents the state of the computer, P represents the power consumption and H describes the heat-up of the environment. The resulting propositional formula is $\varphi = C \rightarrow P \wedge H$.

There is a convenient notation to relate interpretations to formulae: An interpretation I is called a *model* of φ, written as $I \models \varphi$, iff φ is true with regard to I.

Example 2.6. The interpretation $I_0 : P = tt, C = tt, H = tt$ is a model of the formula φ from Example 2.5 while $I_1 : P = tt, C = ff, H = ff$ is not.

Another perspective is to evaluate a given formula φ regarding all its interpretations: A propositional formula φ is called *satisfiable* iff there is at least one interpretation I such that $I \models \varphi$. Otherwise, I is called *unsatisfiable*.

Example 2.7. The formula φ from Example 2.5 is satisfiable (a model is given in Example 2.6). In contrast, the formula $\psi = x \wedge \neg x$ over $V = \{x\}$ is unsatisfiable because the variable x cannot be tt and ff at once.

The *satisfiability problem* for propositional logic (SAT) is defined as follows:

- Given: Formula φ
- Result: *true* iff φ is satisfiable, *false* otherwise

In principle, SAT instances may be solved by exhaustively checking all possible interpretations. But because the number of these interpretations is $2^{|V|}$ and the number of variables $|V|$ may be very large, such a brute-force approach can often not be applied. Instead, algorithms such as *DPLL* [15] or *Conflict-Driven Clause Learning (CDCL)* [31] are used. These algorithms show a significant average performance improvement. However, SAT is *NP-complete* [14], which means that no algorithm is known that efficiently finds a solution for all propositional formulae (and if $P \neq NP$ holds, there is no such algorithm). A tool that implements an approach to decide the satisfiability of propositional formulae is called *SAT solver*.

The next section covers an extension of propositional logic that allows to use variables with other domains.

2.3 Satisfiability Modulo Theories

To express complex properties, it is often required to establish statements about variables with a non-boolean domain.

The concept of *Satisfiability Modulo Theories (SMT)* [7] extends propositional formulae with so-called *theories* that define custom datatypes together with operations on these types. Just like propositional formulae, an SMT formula consists of (one or more) layers of boolean operators. The atoms of a propositional formula (tt, ff, propositional variables) may be replaced by theory atoms. Theory atoms are built with the symbols from the underlying theory and may contain variables with a non-boolean domain. However, the theory atoms as a whole must be of boolean type.

Just like propositional variables, a value needs to be assigned to all variables in order to say that a formula is *true* or *false*. All terms defined in Section 2.2 (interpretation, model, satisfiability) are similarly applied to SMT formulae. The set of the possible values that a variable v can take is called the *domain of* v, denoted as $dom(v)$.

Example 2.8. The statement $x < y \vee z * 4 = 2$ (with $dom(x) = dom(y) = dom(z) = \mathbb{R}$) is an SMT formula in the theory of linear real arithmetic. The interpretation $x = 2, y = 5, z = \frac{1}{2}$ is a model of this formula.

Note that SMT is *many-sorted*, i.e., variables with multiple different types may occur in a formula. To convert values of one type to another type, *cast operators* need to be defined in the theories.

Example 2.9. The formula $(n + m > p) \wedge x < y \wedge \neg b$ (with $dom(n) = dom(m) = dom(p) = \mathbb{R}$ and $dom(x) = dom(y) = \mathbb{N}$) contains three real-valued, two integer and one propositional variable. To directly relate the variables p and x (e.g., comparing them) a cast operator between real and integer values is required.

Solving SMT formulae in a brute-force manner is impossible in general, because the variables may range over infinite domains. Several approaches have been developed to check satisfiability of such formulae: *Eager SMT* (sometimes referred to as *bit-blasting*) approaches produce a propositional formula that is (at least) equi-satisfiable to the original SMT formula and check its satisfiability using a SAT solver. Mostly, the construction of the SAT formula allows the calculation of SMT models directly from the SAT models.

Due to the massive improvements in the performance of SAT solvers, this principle can be applied to many theories, e.g. fixed-size bitvectors. If the theory contains types with an infinite domain, however, such a transformation cannot be performed in general.

Another approach is called *Lazy SMT*. It combines a SAT solver with one or more theory solvers that can handle conjunctions of theory atoms. The idea of Lazy SMT is to replace all theory atoms by propositional variables. The satisfiability of the resulting SAT formula is then checked using a SAT solver. Because this formula is an over-approximation of the original formula, the SAT solver may produce a model that implies inconsistencies between the theory atoms. The theory solver is used to check for such inconsistencies; it produces explanations that can be used to refine the SAT formula. This principle is often referred to as DPLL(T) or CDCL(T). In contrast to Eager SMT, it can also be applied to types with infinite domains [30]. To improve Lazy SMT approaches, the SAT solver may pass partial solutions to the theory solver, which can help to identify inconsistencies between theory atoms faster.

The next section describes the SMT-LIB standard that allows the formalization of SMT statements and presents the theories to be used in this thesis.

2.4 SMT-LIB Standard

During the last decades, many tools have been developed to examine the satisfiability of SMT formulae [20]. For these tools, several different input languages have been specified. This makes comparing these tools a complex task, because for the problem to be solved, it must be translated into the different input languages.

The SMT-LIB Standard [3] aims at the standardization of the input such tools accept and the output they produce. It is maintained by the SMT-LIB initiative and supported by many SMT solvers. The Satisfiability Modulo Theories Competition (SMT-Comp) uses SMT-LIB as input format [2]. The first version was published on July 26th, 2004. This thesis is built upon the newest version 2.6 [4].

The core theory is the foundation of all other theories [33]. It defines the Bool type together with boolean operations and (in)equality operators that all other theories can use to build complex formulae from theory atoms. Together with the standard, the SMT-LIB initiative publishes many different theories for, e.g., reals, integers and strings [5], together with so-called *logics* that may combine different theories and introduce quantifiers.[1]

[1] SMT-LIB contains the quantifiers *forall* and *exists*, just like first-order logic. However, the problems to solve as well as the concepts described in this thesis do not use any quantifiers, so they are not mentioned further.

2.4.1 From SMT to SMT-LIB

This section describes how to express given SMT formulae in the SMT-LIB format. For the translation from SMT to SMT-LIB, it is assumed without loss of generality that the formula consists of a conjunction of arbitrary SMT formulae, i.e., $\varphi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n$. [2]

At first, the logic to use during the SMT-LIB script must be set using the `set-logic` instruction. If only elements from the *Core* theory are used, the logic may be set to any logic, for example *QF_BVFP* (quantifier-free theory of bit-vectors and floating-point, see Section 2.4.2).

After that, the subformulae $\psi_1, \psi_2, \ldots, \psi_n$ are translated to the SMT-LIB format as follows: All theory atoms need to be expressed in the SMT-LIB theories. All variables used in the subformula are declared using the `declare-fun` statement. The theory atoms are connected using the `and`, `or` and `not` operator from the *Core* theory that correspond to the boolean operators defined above. The instruction for the complete subformula is now included into an SMT-LIB `assert` statement. The set of the resulting assert statements is called *satisfiable* if an interpretation exists that is a model of each subformula $\psi_1, \psi_2, \ldots, \psi_n$ and *unsatisfiable* if not. Obviously, the original formula is equi-satisfiable to the resulting set of `assert` statements. As the last step, a `(check-sat)` command instructs the solver to check the satisfiability of the given constraints. After finishing this check, the solver outputs `sat` if the set of `assert` statements is satisfiable and `unsat` if not. If the solver identifies the given instance as satisfiable, a model may be requested using the `(get-model)` statement. In order to do that, the model creation must be enabled at the beginning of the script using the `(set-option :produce-models true)` statement. Finally, the solver is terminated using the `(exit)` command.

Example 2.10. Consider the formula $\varphi = (a \vee (b \wedge c)) \wedge (c \rightarrow d)$. The algorithm described above produces the following SMT-LIB instructions:

```
(set-logic QF_BVFP)
(set-option :produce-models true)
(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(declare-fun d () Bool)
(assert (or a (and b c)))
(assert (=> c d))
(check-sat)
(get-model)
(exit)
```

In addition to that, SMT-LIB allows to create sets of instructions interactively: All instructions are added to an *assertion stack*. A layer may be added to this stack using the `(push 1)` and removed using the `(pop 1)` instruction. The set of the constraints the solver checks contains all instructions from any layer of the assertion stack. This is very useful to run Bounded Model Checking and k-Induction on SMT solvers, which is described in Section 2.6 and Section 2.7.

2.4.2 The Logic QF_BVFP

This thesis is built on the logic QF_BVFP (quantifier-free logic of bit-vectors and floating-point), as this is the natural logic for model checking on C programs. In this section, a fundamental introduction to this logic is provided. `BitVecs` in SMT-LIB are fixed-size bit fields which can be

[2] Note that φ does **not** need to be in conjunctive normal form.

used to model integers of any bitwidth in C. There exist different ways to define bit-vectors in SMT-LIB. In this thesis, they are created from binary or hexadecimal representations.

Example 2.11. The SMT-LIB instructions

```
(declare-fun x () (_ BitVec 32))
(assert (= x #xffffffff))
```

declare a 32-bit `BitVec` variable `x` and assert that it has the value `0xffffffff` (-1 as signed integer, $2^{32}-1$ as unsigned integer).

Unlike C, SMT-LIB does not distinguish between signed and unsigned integers. Operators for both signed and unsigned operations are provided, but the bit-vectors itself cannot be defined as signed or unsigned; instead, signed and unsigned operators may be applied to *any* bitvectors. An important limitation of bit-vectors in SMT-LIB is that any binary or ternary operation must be performed on BitVecs of the same length. To ensure that, SMT-LIB has the `concat`, `sign-extend`, `zero_extend` and `extract` operation. Other operators on `BitVecs` are not presented here as they are not relevant to the approach described in this thesis. They may be found in [25].

`FloatingPoint` values, on the other hand, follow the IEEE 754 standard [18] and model floating-point values in C. As for `BitVecs`, `FP` values of any length may be created. When declaring them, the number of exponent bits and the number of mantissa bits (including the hidden bit) must be specified. The shorthands for the common floating-point formats `half`, `float`, `double`, and `quadruple` are `Float16`, `Float32`, `Float64`, and `Float128`. The value of `FloatingPoint` variables can be set using a single `BitVec` representing the complete floating-point value or using separate `BitVecs` for sign, exponent and mantissa.

Example 2.12. A `float` variable in C (32 bit) may be declared as follows:

```
(declare-fun x () Float32)
```

To assert that this variable has the value `2.0f`, the following instructions can be used:

```
(assert (= x (fp #b0 #x80 #b00000000000000000000000)))
```

The `FloatingPoint` theory defines a `fp.eq` resp. `fp.ne` operator. They express the comparison operation for floating-point values, which means that `(fp.eq f1 f2)` is `true` if f1 is $+0$ and f2 is -0 (or vice versa) and `false` if at least one of them is NaN. In contrast, the `=` operator from the Core theory models the bit-wise equivalence between its arguments. As for `BitVecs`, other `FloatingPoint` operators are not mentioned here because they are not used along with this thesis. A complete overview of the `FloatingPoint` theory may be found in [34].

2.5 From C Programs to SMT-LIB Scripts

The previous sections describe how the SMT-LIB format can be used to perform model checking of SMT formulae. However, it was not described how the set of SMT-LIB instructions is created.

Many state-of-the-art embedded systems are developed using the programming language C. This language, however, is not suitable for direct translation into the SMT-LIB format: C is a imperative programming language; a C program clearly defines *what* the computer should do and *how*. SMT-LIB, on the other hand, is a language to specify a set of conditions and check whether this set can be fulfilled. One might say that SMT-LIB defines the problem to solve, but not *how* to solve it. Additionally, C programs contain some features such as `while` loops and functions that cannot be expressed in SMT-LIB.

In order to translate C programs to SMT-LIB instructions automatically, several transformations must be applied. This section describes these transformations and mentions the restrictions that C programs to be verified must observe.

C programs for embedded systems contain three types of variables: *State variables* are used to represent the internal state of the embedded system. They may be declared as global variables; to define the initial state of the system, a value must be assigned to all state variables. During the execution of the embedded system, this value may change. To represent the inputs coming from sensors, switches, etc., *input variables* are used. Because they introduce external values into the system, values cannot be assigned to them. Input variables can be declared as global variables (without being initialized), or as formal variables (arguments of a function). Finally, *output variables* can be defined to export values to, e.g., actuators or other embedded systems. The transition from one step to another is defined using a *function*. This function takes all input values as well as all state values and computes the new state of the system from them.

Example 2.13. The source code of an embedded system to verify may look like this:

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

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

    executions++;
}
```

This program defines two state variables `counter` and `executions` and a function `f` that requires an input variable `x`.

To make applying transformations easier, the C programs are converted into an internal format called SMI. This format is also an imperative programming language with similar control flow structures as C (while loops, if statements, etc.), but a different syntax and additional features (e.g., explicit upper / lower bounds for variables). `goto` or `label` statements are not allowed in the C programs (they must be replaced by `while` statements). 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. Finally, some first modifications are applied, for example, `do {...} while (...);` are replaced by `while (...) {...}` statements. State variables may occur primed (e.g. `x'`) or unprimed (e.g. `x`) in the transition relation of a SMI program. Unprimed variables represent the "old" value of the state variable relation while primed variables represent the "new" (updated) value of this variable.

The SMI representation is now used to perform several other transformations: All function calls are replaced by the definition of the functions; this step is often referred to as *inlining*. This requires that the C programs to verify do not contain recursive functions. Furthermore, all complex data structures are flattened, i.e. replaced by single integer / floating-point variables. As a next step, the property is injected into the SMI program. In the context of BTC Embedded*Platform*®, the properties to check are essentially reachability constraints: Given a C program and a specific instruction from it, is there a sequence of input data so that the execution hits this particular instruction? Other types of properties (e.g., functional properties that constrain the values of specific variables) are internally reduced to reachability problems. There is no dedicated

`property` statement in SMI programs. Instead, all verification tasks contain a specific variable `invariance_property`. This variable is initialized with value 1. In the transition relation, this variable is updated such that `invariance_property` is 0 iff the original property is violated in the same situation.

Example 2.14. The property `counter' < executions'` for the program from Example 2.13 can be expressed using the `invariance_property` variable as follows:

```
invariance_property' = (counter' < executions');
```

After some further transformations, the SMI program is now ready for another very important step: All remaining `while` and `for` loops are unrolled, i.e., a loop that may run at most n times is replaced by n copies of the loop body (with only the loop variable changing in each copy). This requires an upper limit for the number of loop runs. For simple loops such as `for(i = 0; i < N; i++)` with `N` being a constant, this upper limit can be concluded. If that is not possible, a user-defined number of loop unrollings is used and additional variables are introduced to detect an insufficient number of loop unrollings. In this case, the transformation does not necessarily produce a semantically equivalent program.

Example 2.15. The loop skeleton

```
while (cond) {
    BODY
}
```

would be unrolled as follows, using a number of loop unrollings of 3:

```
insufficient_number_of_loop_unrollings = false;
if (cond) {
    COPY OF BODY 1
    if (cond) {
        COPY OF BODY 2
        if (cond) {
            COPY OF BODY 3
            if (cond) {
                insufficient_number_of_loop_unrollings = true;
            }
        }
    }
}
```

If `insuffient_number_of_loop_unrollings` is `true` in a model being found, the model may be invalid because the number of loop unrollings may be too small.

The only loop that remains is a `while (1)` loop that represents the transition from one step to another. Moreover, `if` statements are replaced by the `? :` ternary operator. The last important transformation is to convert the SMI program into *static single assignment (SSA)* form: It is ensured that each variable is assigned *exactly once* in a program. Furthermore, for each access to the variable, it is mentioned whether the value from the previous or the current step shall be used. If necessary, further variables are introduced to accomplish SSA form. The advantage of the SSA form is that the program is now *commutative*: Two instructions may be swapped without changing the semantics of the program. The SMI program now consists of a set of assignments to the variables in step 0 (called *initialization*), together with a `while (1)` loop that contains a

set of assignments for the next-step variables (called *transition*). This format is called *elaborated SMI*. The initialization, as well as the transition for a particular step, can be interpreted as SMT formulae.

Example 2.16. The program from Example 2.13 together with the property "the value of the variable `counter` is smaller than the value of the variable `executions`" now has the following structure (C syntax is used here instead of SMI syntax because most readers will be more familiar with it):

```
// INIT section

int counter = 0;
int executions = 0;
int x;

// TRANS section
while(1)
{
    counter'             = (x != 0 ? counter + x : 0);
    executions'          = executions + 1;
    invariance_property' = (counter' < executions');
}
```

The step-wise unrolling of the `while(1)` loop is the basic idea of Bounded Model Checking which will be explained in Section 2.6.

At this point, the SMI program can be translated into the several formats required by the various model checkers being used. For SMT-LIB and some other model checkers, this conversion is performed by the tool `smi2engine`.

As already mentioned, SMT-LIB is not able to express `while` loops. Because of this limitation, the loop needs to be unrolled for each transition step. In order to support this feature, `smi2engine` generates five different files containing general function definitions, variable declaration, initialization, transition and property. The declaration, transition, and property file contain placeholders and are instantiated for each concrete step.

Example 2.17. `smi2engine` generates the following SMT-LIB instructions for the program in Example 2.13:

```
; 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)
```

`$$old` and `$$new` are later replaced by the concrete step numbers.
The complete files that smi2engine generates can be found in Appendix A.

The tool `smi_smt2` is responsible for instantiating these files for each step. Furthermore, it provides them to the solver process and evaluates the result of each `(check-sat)` statement. For communication, pipes are connected to the standard input and output of the solver process.

The next sections introduce techniques to efficiently perform model checking on systems with the described layout.

2.6 Bounded Model Checking

In Section 2.5, the structure of the C source code and the properties to investigate have been specified. This section describes how to perform model checking on such systems.

How can a software developer A convince another developer B that the source code B wrote does not fulfill a specific (reachability) requirement? An *execution trace* must be provided that leads to a "bad" state of the system (i.e., a state violating the requirement). The execution trace must fulfill several conditions to suffice as such a proof:

- The first state of the execution trace must be compatible with the initialization, i.e., all state variables are set to the value demanded by the initialization.
- Any other state results from the application of the transition to the values from the previous step, together with the current input values.
- The values of the variables in the last step must violate the property.

Example 2.18. In order to show that the C program from Example 2.13 does not fulfill a given requirement, n values must be given for each state variable (`counter` and `executions`) such that

- in step 0, both variables have the value 0
- the values in each step result from the values in the previous step
- the values of these variables in the last step violate the requirement

Bounded Model Checking (BMC) [9] [12] is a model checking algorithm that is widely used for the verification of developed systems. This technique was originally developed for the formal verification of hardware circuits, but can be applied to software verification as well. It is an alternative approach to *Binary Decision Diagrams (BDDs)* that have previously been used for such verification tasks [10]. The main idea of Bounded Model Checking is to check the validity of a property P in a system consisting of an initialization I and a transition relation T with respect to a fixed number of steps $k \in \mathbb{N}$. By letting k start at the value 0 and increasing it stepwise,

the validity of the property can be guaranteed for any finite number of steps. When referring to the concrete instantiation of the transition and property files for a concrete step n, the notations $T^{(n,n+1)}$ and $P^{(n)}$ are used.

Verifying that the property holds after 0 steps (the BMC_0 check) is fairly simple in the original BMC approach: A violation after 0 steps means that the initialization itself violates the property.

Example 2.19. In the C program from Example 2.13, the two variables `counter` and `executions` are both initialized with the value 0. The following property is violated for this program in 0 steps: $counter \neq 0 \vee executions \neq 0$.

An execution trace that shows such a violation must be conformant with the initialization while violating the property. In other words, a SMT model checker can check for such an execution trace by solving the formula $I \wedge \neg P^{(0)}$. In the context of this thesis, SMT-LIB conformant solvers are used. As a preparation step, the several parts of the C program (initialization, transition) and the property to be checked must first be translated into SMT-LIB instructions - a job that the toolchain described in Section 2.5 performs. As mentioned above, two other files – the header and declaration file – are created for technical reasons.

This particular encoding has an important limitation: The definition of the property is provided in the transition only, by assigning a value to the special variable `invariance_property`. The property statement itself only asserts that `invariance_property` is `true`. This implies that the BMC_0 formula is *always* unsatisfiable with this encoding – it contains the two contradictory statements `(assert invariance_property$$0)` (the initialization of the variable) and `(assert (not invariance_property$$0))` (the property statement).

Example 2.20. An SMT-LIB solver can use the following script to perform the BMC_0 check for the program and property in Example 2.13 (as mentioned above, the solver will always return `unsat` for these checks):

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

; DECLARATION FOR STEP 0
(declare-fun counter$$0 () (_ BitVec 32))
(declare-fun executions$$0 () (_ BitVec 32))
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun invariance_property$$0 () Bool)

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

; NEGATED PROPERTY FOR STEP 0
(assert (not invariance_property$$0))

; check the satisfiability of the given constraints
(check-sat)
```

BMC_0 can be used to uncover violations of the property in step 0. Furthermore, the ideas described above can be extended to further steps.

Example 2.21. Consider the property `counter' > executions'`. This requirement cannot be violated in step 0.

How can a violation in a later step n be detected using a model checker? As for BMC_0, an execution trace that shows such a violation must be consistent with the initialization. Furthermore, the variable values in two consecutive steps must be conformant with the transition relation. Finally, the variable values in step n must violate the property. A model checker can find such an execution trace by checking the satisfiability of the formula $I \wedge T^{(0,1)} \wedge T^{(1,2)} \wedge \cdots \wedge T^{(n-1,n)} \wedge \neg P^{(n)}$. The check that needs to be done for step n is denoted as BMC_n. In contrast to BMC_0, the particular encoding does not affect the function of all other BMC steps, because the transition is applied for each step (except the first), which includes the full definition of the property.

Example 2.22. The following SMT-LIB script can be used to perform the BMC_1 check for the program in Example 2.13, together with the property from Example 2.21:

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

; DECLARATION FOR STEP 0
(declare-fun counter$$0 () (_ BitVec 32))
(declare-fun executions$$0 () (_ BitVec 32))
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun invariance_property$$0 () Bool)

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

; DECLARATION FOR STEP 1
(declare-fun counter$$1 () (_ BitVec 32))
(declare-fun executions$$1 () (_ BitVec 32))
(declare-fun x$$1 () (_ BitVec 32))
(declare-fun invariance_property$$1 () Bool)

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= counter$$1 (ite (distinct x$$0 #x00000000)
                      (bvadd counter$$0 x$$0) #x00000000)))
(assert (= executions$$1 (bvadd executions$$0 #x00000001)))
(assert (= invariance_property$$1 (bvsge counter$$1 executions$$1)))

; NEGATED PROPERTY FOR STEP 1
(assert (not invariance_property$$1))

; check the satisfiability of the given constraints
(check-sat)
```

The solver may return the model in Table 1.

With the BMC_k approach, any violation within a finite number of steps can be detected. However, for any given finite number of steps n, a property can be given that is not violated in all steps $0, 1, \ldots, n-1$, but in the depth n.

Variable Name	Depth 0	Depth 1
counter	0	2
executions	0	1
invariance_property	true	false

Table 1: A model of the above SMT-LIB script

Example 2.23. Consider the program in Example 2.13, together with an arbitrary $n \in \mathbb{N}$. The property

$$executions < n$$

is fulfilled by the program in the steps $0, 1, \ldots, n-1$, because the variable `counter` is initialized with value 0 and incremented in each step. However, it can easily be concluded that the property is violated in step n.

In order to limit the runtime of the BMC approach, several BMC algorithms use a user-defined maximum number of steps. If this number of steps is exceeded without a violation being found, no definite conclusion can be made because the number of steps may be too low to get a violation. Another consequence of this problem is that the conformance of a program to a given property cannot be proven by BMC: To do so, one would have to perform a BMC_n check for *all* $n \in \mathbb{N}$, which is impossible in finite time.

Example 2.24. In the program from Example 2.13, consider the proposition that the variable `counter` always changes its value:

$$executions' \neq executions$$

This property is never violated and all BMC_n formulae are unsatisfiable, but BMC is unable to detect this.

On the other hand, BMC is able to detect all violations that may occur at a given depth n. Therefore, BMC is a semi-decision procedure for the problem

Can the given property be violated during any execution of the given program?

The next section describes an approach to prove that a program conforms to a given property.

2.7 k-Induction

This section provides a complementary approach to BMC called k-Induction [16]. It is able to detect that *all* BMC_n formulae with n being greater than a particular number of steps k are unsatisfiable. As a preparation, consider the BMC_n formulae from the last section:

$$\begin{aligned}
&\text{BMC}_0: && I \wedge \neg P^{(0)} \\
&\text{BMC}_1: && I \wedge T^{(0,1)} \wedge \neg P^{(1)} \\
&\text{BMC}_2: && I \wedge T^{(0,1)} \wedge T^{(1,2)} \wedge \neg P^{(2)} \\
&\ldots \\
&\text{BMC}_n: && I \wedge \textstyle\bigwedge_{i=1}^{n} (T^{(i-1,i)}) \wedge \neg P^{(n)}
\end{aligned}$$

If the formula $I \wedge \neg P^{(0)}$ is unsatisfiable, there is no possibility to violate the property in the 0th step. This implies that the semantics of the BMC_n formulae do not change if $P^{(0)}$ is added to

them. For the same reason, each term $P^{(k)}$ can be added to all BMC_n formulae with $n > k$, which results in the new formulae BMC'_k:

$$\begin{aligned}
\text{BMC}'_0 &: \quad I \wedge \neg P^{(0)} \\
\text{BMC}'_1 &: \quad I \wedge P^{(0)} \wedge T^{(0,1)} \wedge \neg P^{(1)} \\
\text{BMC}'_2 &: \quad I \wedge P^{(0)} \wedge T^{(0,1)} \wedge P^{(1)} \wedge T^{(1,2)} \wedge \neg P^{(2)} \\
&\dots \\
\text{BMC}'_n &: \quad I \wedge \textstyle\bigwedge_{i=1}^{n} (P^{(i-1)} \wedge T^{(i-1,i)}) \wedge \neg P^{(n)}
\end{aligned}$$

It is noticeable that all the BMC'_n formulae with $n \leq 1$ contain the suffix $P^{(n-1)} \wedge T^{(n-1,n)} \wedge \neg P^{(n)}$. This suffix encodes that the property is fulfilled in the penultimate, but violated in the last step. Furthermore, the parameter n is only used to name the variables in the formula parts P and T and has no impact on the formulae itself. Because of that, all of these suffixes are equi-satisfiable to the formula $P^{(0)} \wedge T^{(0,1)} \wedge \neg P^{(1)}$, which is called IND_1. The satisfiability of this formula can be checked, being referred to as *1-induction*. If this formula is unsatisfiable, *all* formulae BMC_n with $n \geq 1$ are also unsatisfiable.

Again, the particular encoding used in this thesis has an influence on the effectiveness of 1-induction, because the full definition of the property can be added as a part of a transition step only. Instead of requiring the first step to fulfill all property constraints, it only demands that the variable `invariance_property` is `true` in the first step. This condition is fulfilled by more states of the system, which implies that this formula will be satisfiable more often than the original k-Induction formula.

Example 2.25. Consider the program from Example 2.13, together with the property from Example 2.24. In order to prove that this formula is unsatisfiable, an SMT-LIB model checker can solve the following script:

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

; DECLARATION FOR STEP 0
(declare-fun counter$$0 () (_ BitVec 32))
(declare-fun executions$$0 () (_ BitVec 32))
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun invariance_property$$0 () Bool)

; PROPERTY FOR STEP 0
(assert invariance_property$$0)

; DECLARATION FOR STEP 1
(declare-fun counter$$1 () (_ BitVec 32))
(declare-fun executions$$1 () (_ BitVec 32))
(declare-fun x$$1 () (_ BitVec 32))
(declare-fun invariance_property$$1 () Bool)

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= counter$$1 (ite (distinct x$$1 #x00000000)
                      (bvadd counter$$0 x$$1) #x00000000)))
(assert (= executions$$1 (bvadd executions$$0 #x00000001)))
(assert (= invariance_property$$1 (distinct executions$$1 executions$$0)))
```

```
; NEGATED PROPERTY FOR STEP 1
(assert (not invariance_property$$1))

; check the satisfiability of the given constraints
(check-sat)
```

The model checker finds out that this formula is unsatisfiable.

The same principle can be applied to longer suffixes: In general, all BMC_n formulae with $n > k$ share the same suffix $\bigwedge_{i=1}^{k}(P^{(i-1)} \wedge T^{(i-1,i)}) \wedge \neg P^{(k)}$. This formula is called IND_k; successively checking the satisfiability of the formulae IND_k for all $k \in \mathbb{N}$ is called *k-Induction*.

As for the first induction steps, the encoding being used weakens the k-Induction formulae. Many programs require more induction steps with this encoding than with the original one. This is the main reason why in BTC Embedded*Platform*®, k-Induction is often performed in step 2, although the property can be shown to be valid by 1-induction using other encodings.

Note that IND_k can only be used to prove the unsatisfiability of those BMC_n with $n > k$: If a formula IND_k is unsatisfiable, a BMC_n formula with $n \leq k$ can still be satisfiable. Because of that, BMC and k-Induction must be combined in order to check the conformance of a program with a property:

1. Solve the BMC_0 formula. If this formula is satisfiable, get a model from the solver and terminate (violation found).
2. Set `i` to 1.
3. Solve the IND_i formula. If this formula is unsatisfiable, terminate (property proven).
4. Solve the BMC_i formula. If this formula is satisfiable, get a model from the solver and terminate (violation found).
5. Increment `i` by 1 and go to step 3.

3 Extending k-Induction with Elimination of Constants

The last section has covered two approaches to perform model checking on systems with a specific layout. The main contribution of the following sections is an approach to detect and eliminate *constants*, i.e., variables that keep their initial values during the complete execution of the program, regardless of the incoming inputs. The following section describes why this approach promises improvements in the strength of k-Induction.

3.1 The Problem with k-Induction and How Elimination of Constants can Help

Consider the formula for k-Induction from section Section 2.7:

$$\bigwedge_{i=1}^{k}(P^{(i-1)} \wedge T^{(i-1,i)}) \wedge \neg P^{(k)}$$

It can be seen that the formula contains a property and transition for each except the last step. Furthermore, it can be observed that the initialization is missing. This is necessary because

k-Induction uses inductive reasoning: The validity for the property in each step follows from the conformance to the property of an earlier step. Therefore, no assumptions can be made about this earlier state except that it fulfills the given property.

The SMT-LIB encoding can contain constants, i.e., variables that get assigned a value in the 0th step and keep this value regardless of the incoming input values. Without the information in the initialization, however, these constants can often not be detected: In many situations, the exact initial value is needed in order to show that a variable is a constant.

Example 3.1. Consider the following SMT-LIB instructions:

```
; DECLARATION FOR STEP 0
(declare-fun x$$new () (_ BitVec 32))

; INITIALIZATION FOR STEP 0
(assert (= x$$0 #x00000000))

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= x$$new (ite (= x$$old #x00000000)
                       #x00000000
                       (bvadd x$$old #x00000001))))

; check the satisfiability of the given constraints
(check-sat)
```

`x` is a constant (it keeps the value 0), but without the initialization, a solver has no chance to conclude that.

The idea of this thesis is to preprocess the SMT-LIB scripts in order to find such constants. By providing this knowledge to the solver, it can be used during k-Induction checks.

3.2 Elimination of Constants for Single Variables

This section introduces an algorithm that allows to decide whether a given variable is a constant using SMT-LIB compliant solvers. This algorithm can be used to check a single variable. Later sections describe extensions of this algorithm that allow to check multiple variables at once.

A requirement for this algorithm is that it only identifies variables as constants that have an initial value. While this is true for state variables, it is not the case for input and auxiliary variables: Input variables get their values from outside of the system; assigning an initial value to them would not make any sense. Auxiliary variables are just shorthands for complex expressions in the transition relation and refer only to the variables of the current step. Because the transition relation is not used to calculate the initial values, auxiliary variables do not need such a value. In order to fulfill that requirement, the algorithm sorts out all input and auxiliary variables before performing elimination of constants.

The target of this section is to construct for each state variable v an SMT-LIB script that is satisfiable if v is *not* a constant. When assuming the existence of a correct and complete SMT-LIB solver (i.e., the solver returns unsatisfiable if and only if there is no model for the investigated SMT-LIB script), this script is correct (i.e., if the solver returns `unsat`, then the variable is indeed a constant), but not necessarily complete (i.e., if the solver returns `sat`, the variable may still be a constant). In order to perform such a check, the formula needs to express a value change during a transition step. This implies that the script to check a given variable v with an initial value v_0 must conform to the following general structure:

- Assert that the variable has the value v_0 in step 0.
- Apply the transition relation from step 0 to step 1.
- Assert that the variable has another value $v_1 \neq v_0$.

Example 3.2. Consider the SMT-LIB scripts from Example 3.1. The following SMT-LIB script can be used to prove that the variable `x` is a constant:

```
; DECLARATION FOR STEP 0
(declare-fun x$$0 () (_ BitVec 32))

; x == 0 IN STEP 0
(assert (= x$$0 #x00000000))

; DECLARATION FOR STEP 1
(declare-fun x$$1 () (_ BitVec 32))

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= x$$1 (ite (= x$$0 #x00000000)
                     #x00000000
                     (bvadd x$$0 #x00000001))))

; x != 0 IN STEP 1
(assert (distinct x$$1 #x00000000))

; check the satisfiability of the given constraints
(check-sat)
```

When looking at the definition, it may not be clear why an `unsat` result implies that the corresponding variable is indeed a constant: The main reason of using techniques such as Bounded Model Checking and k-Induction is that there is no general upper bound for the number of steps needed to violate a particular invariant. For the property that a specific variable v cannot change its value, however, it is enough to consider only one transition step. This can be proven by contradiction: Assume that there is a variable v that cannot be detected with the above formula, but with a formula containing more than one transition steps. The model of such a formula contains $n-1$ steps where $v = 0$, followed by one step where $v \neq 0$. When looking at the last two steps, it can be seen that a model can be derived for our one-step-formula: In the penultimate step, $v = v_0$ holds while in the last step, v must have changed its value, which is exactly the constraint our formula establishes. This is a contradiction to our assumption that our original formula does not have a model that shows a value change of the variable v.

This implies that this algorithm is indeed correct, i.e., if the solver returns `unsat` for such a formula, the corresponding variable is indeed a constant, because there exists no model that can show a value change of this variable. On the other hand, this approach is not complete: If the algorithm returns `sat`, it has found a transition from a state where $v = v_0$ holds to a state where $v = v_0$ does not hold. However, it is not checked whether this first state is reachable; If the solver returns `sat`, thus identifying the variable as non-constant, it may still be the case that the first step is unreachable in all models of the created SMT-LIB script.

Example 3.3. Consider the following SMT-LIB benchmark:

```
; DECLARATION FOR STEP n
(declare-fun x$$new () (_ BitVec 32))
(declare-fun y$$new () (_ BitVec 32))
(declare-fun invariance_property$$new () Bool)

; INITIALIZATION
(assert (= x$$0 #x00000000))
(assert (= y$$0 #x00000000))

; TRANSITION FROM STEP n TO STEP n+1
(assert (= x$$new (ite (distinct y$$old #x00000000)
                       #x00000001
                       #x00000000)))
(assert (= y$$new y$$old))
(assert (= invariance_property$$new (bvslt x$$new #x00000002)))

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

It can be seen that the variable `x` is a constant, because `y` is always 0. Using the created SMT-LIB script, however, a solver is not able to deduce this fact (the formula is satisfiable):

```
; DECLARATION FOR STEP 0
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun y$$0 () (_ BitVec 32))
(declare-fun invariance_property$$0 () Bool)

; x == 0 IN STEP 0
(assert (= x$$0 #x00000000))

; DECLARATION FOR STEP 1
(declare-fun x$$1 () (_ BitVec 32))
(declare-fun y$$1 () (_ BitVec 32))
(declare-fun invariance_property$$1 () Bool)

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= x$$1 (ite (distinct y$$0 #x00000000)
                     #x00000001
                     #x00000000)))
(assert (= y$$1 y$$0))
(assert (= invariance_property$$1 (bvslt x$$1 #x00000002)))

; x != 0 IN STEP 1
(assert (distinct x$$1 #x00000000))

; check the satisfiability of the given constraints
(check-sat)
```

After a constant has been detected, this knowledge needs to be passed to the solver. The first idea utilizes the fact that each state variable gets assigned a value in the transition relation.

The corresponding `assert` statement of each constant `x` with value `v` can be replaced as follows: `(assert (= x v))`. Another approach is to leave the original assignment unchanged and add this statement to the transition relation: An assignment to a specific variable can also have impact on other variables used in the definition of this variable. In order to allow the solver to use this knowledge, the original assignment must also be present in the updated transition relation.

An important disadvantage of both approaches is that the variable `x` is still part of the transition relation: While the solver can easily conclude the value of `x` in each step, it cannot conclude that `x` has the same value in all steps, because the solver has no direct understanding of the BMC specific formula structure. As a consequence, the instantiations of `x` for each step must be handled like any other variable. Because of that, another approach is used during this thesis: All occurrences of a constant `x` are replaced by the constant value. Furthermore, the declarations of these constants are removed. After doing this, the variable is no longer contained in the benchmarks. Experimental evaluation during this thesis has proven this strategy to perform much better than the other approaches.

If the knowledge about constants is passed to the solver, the above procedure may be able to identify further variables that cannot change their value.

Example 3.4. Consider the following C program:

```
// INIT section

int x = 0;
int y = 1;

// TRANS section
while(1)
{
    x' = (x != 0 ? x + 1 : 0);
    y' = x';
}
```

If the variable `y` is checked first, the solver returns `sat`, together with the following model:
`x$$0 = 1, y$$0 = 1, x$$1 = 2, y$$1 = 2`
In order to prove that `y` is indeed a constant, the variable `x` must be considered. After that, the solver can return to variable `y` and show that it cannot change its value.

Because of that, the algorithm presented above must be repeated until the set of constants reaches a fixed-point, i.e., an iteration with no new constants being identified. The user may also set a timeout for the whole procedure.

The next sections describe extensions of this algorithm that allow to check multiple variables at once.

3.3 Disjunctive Elimination of Constants

The algorithm presented in the previous section may need a significant amount of time: Just like for Bounded Model Checking or k-Induction, no general upper limit can be given for the complexity of the resulting formula and for the execution time. Despite techniques such as parallel solving (using multiple solver instances to perform parallel elimination of constants), improved encodings can be used to speed up this process. In this section, an approach is described that builds upon the basic elimination scheme from the last section, but extends it to check multiple variables using one `(check-sat)` invocation. To ensure that all of these variables start with their initial value, an

`assert` statement is added for each of them. The important idea in this approach is to take all separate statements that assert a value change of the variables and combine them using boolean `or` operators.

Example 3.5. The following formula is created for the variables in Example 3.4:

```
; DECLARATION FOR STEP 0
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun y$$0 () (_ BitVec 32))

; x == 0 IN STEP 0
(assert (= x$$0 #x00000000))

; y == 0 IN STEP 0
(assert (= y$$0 #x00000000))

; DECLARATION FOR STEP 1
(declare-fun x$$1 () (_ BitVec 32))
(declare-fun y$$1 () (_ BitVec 32))

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= x$$1 (ite (distinct x$$0 #x00000000)
                     (bvadd x$$0 #x00000001)
                     #x00000000)))
(assert (= y$$1 x$$1))

; x != 0 or y != 0 IN STEP 1
(assert (or (distinct x$$1 #x00000000) (distinct y$$1 #x00000000)))

; check the satisfiability of the given constraints
(check-sat)
```

If this formula is satisfiable, there must be at least one variable for which the conjunction of the two given constraints (for step 0 and step 1) is satisfiable, which means that the associated variable has changed its value. By investigating the model given by the solver, all such variables can be identified; the procedure can then be applied to the remaining variables. If the formula is unsatisfiable, no such variable can exist; all investigated variables must keep their initial values.

In the implementation for this thesis, the algorithm starts with the complete set of variables. It repeats the above procedure until the formula is unsatisfiable or no variables are left. As for the elimination of a single variable, this procedure must be repeated until the set of constants reaches a fixed point or until a user-given timeout is reached.

The next section introduces another algorithm to check multiple variables at once.

3.4 Conjunctive Elimination of Constants

In the last section, the `(distinct)` statements of multiple variables have been combined using the `(or)` operator. Another possibility is to connect them using `(and)` operators.

Example 3.6. For the Example 3.4, the following formula is built:

```
; DECLARATION FOR STEP 0
(declare-fun x$$0 () (_ BitVec 32))
(declare-fun y$$0 () (_ BitVec 32))

; x == 0 IN STEP 0
(assert (= x$$0 #x00000000))

; y == 0 IN STEP 0
(assert (= y$$0 #x00000000))

; DECLARATION FOR STEP 1
(declare-fun x$$1 () (_ BitVec 32))
(declare-fun y$$1 () (_ BitVec 32))

; TRANSITION FROM STEP 0 TO STEP 1
(assert (= x$$1 (ite (distinct x$$0 #x00000000)
                     (bvadd x$$0 #x00000001)
                     #x00000000)))
(assert (= y$$1 x$$1))

; x != 0 and y != 0 IN STEP 1
(assert (and (distinct x$$1 #x00000000) (distinct y$$1 #x00000000)))

; check the satisfiability of the given constraints
(check-sat)
```

The conjunction of `(distinct)` is satisfiable only if there is a model that makes *all* conjuncts evaluate to *tt*. This implies that this model encodes a value change of all variables. If the formula consists only of one variable and is still unsatisfiable, this variable is shown to be a constant.

The main idea is to start with N disjunctive sets of variables and split them recursively until the parts are satisfiable or until only one variable is left. The strategy to identify the original set of variables can have a significant impact on the runtime this algorithm has (see Section 4).

The simplest approach is to start with the complete set of variables. However, this idea does not incorporate any knowledge about the benchmark's internal structure. A more sophisticated approach makes use of the *dependency graph* that contains all variables as nodes and an edge from v to w iff w directly depends on v. The subsets to be evaluated by conjunctive elimination are created using the Strongly Connected Components (SCCs) of this graph: A Strongly Connected Component is a subset V_i of all vertices in the dependency graph such that each node v is reachable from each node w in V_i. To identify the SCCs, the Tarjan Algorithm [32] can be used.

In addition to the normal dependency graph, this thesis also considers SCCs on the inverse dependency graph, i.e., the graph that contains an edge from v to w iff v directly depends on w. The reason is that the edges in the reverse dependency graph are more evenly distributed: For the definition of a variable, only three different variables are used. This means that in the reverse dependency graph, only three edges may start at each node. On the other hand, each variable may be used to define an unlimited number of other variables. Thus, there is no upper limit for the number of edges in the normal dependency graph that start at a given node. Because of that, it may happen that either the normal or the inverse dependency graph perform better in finding subsets of variables to check.

The next sections describe the experimental results that have been achieved with the several elimination strategies.

4 Experimental Evaluation

In the previous sections, strategies for the detection of constants have been proposed. This section deals with the evaluation of these strategies on a concrete benchmark set. The influence of several detection settings is evaluated concerning the runtime needed and the depth where a result was gained. Finally, tables are provided for each solver that summarize the results. As a preparation, the used benchmark set is introduced.

4.1 The Benchmark Set

For the evaluation of the several approaches presented in this thesis, a set of 8778 benchmarks is used. This benchmark set has been derived from original customer tasks and serves as an entrance point to judge the performance of backend solvers on typical BTC Embedded*Platform*® use-cases. As mentioned in Section 2.5, the model checking toolchain of BTC Embedded*Platform*® allows the inspection of reachability properties. The Embedded*Platform*® offers three different use-cases that are internally solved by reducing them to reachability problems: CCC Checks (investigating the consistency, correctness, and completeness of given properties), Formal Verification (checking the validity of a user-given property), and Automatic Test Case Generation. While the first two play a subordinate role for this thesis, the latter one is very important to understand the origin of the benchmark set.

In today's software development, *code coverage* plays an important role. By guarding written software with tests, the quality of the outcoming product can be improved significantly. This has also influenced several software development standards, such as the ISO 26262 standard [19], which demands the detection and elimination of so-called *dead code*, i.e., instructions or branches in the control flow that can never be reached during the execution of the program.

While ISO 26262 suggests metrics to measure code coverage, it does not specify how the needed test cases should be created. Instead of letting the user define these test cases, BTC Embedded-Platform uses a combination of randomization techniques (comparable to *fuzzing* approaches) and model checking to gain the required test cases. At first, random inputs are generated to cover most of the statements. For each remaining statement (or branch), a corresponding reachability property is created and investigated using the model checking toolchain. If the model checker finds a model for the resulting formula, a test case has been identified that can be added to the existing set of test cases. Otherwise, the investigated statement or branch is shown to be unreachable and can safely be removed from the source code.

The benchmark set used in this thesis was created by applying this test case generation to 11 C projects. These were derived from Simulink® models with the help of TargetLink®. The random engine has been turned off, so that verification tasks have been created for all branches in the source code. The resulting SMI programs have been stored; this allows to use them directly as input for the core tools developed for the SMT-LIB format.

The number of benchmarks gained from the several C projects is shown in table Table 2. It can be seen that the set model_08 contains more than half of the benchmarks. The benchmarks contain various bit-vector or floating-point instructions; details may be found in [22].

The next section introduces the backend solvers to use and presents the results gained without the ideas in this thesis.

4.2 SMT-LIB Backend Solvers and Prior Results

The SMT-LIB format described in Section 2.4 is only a standard for SMT formulae specification. For solving concrete formulae, *solvers* are needed that support this standard. Obviously, a solver to

Name	Number of benchmarks
model_02	22
model_03	15
model_04	819
model_06	10
model_08	5159
model_09	631
model_10	4
model_11	174
model_12	1577
model_13	357
model_14	11

Table 2: The number of benchmarks per C project.

be used during this thesis must be able to solve formulae of the `QF_BVFP` theories. Three important solvers that fulfill this requirement are CVC4 [6], MathSAT5 [11] and Z3 [24]. All of these solvers use *eager SMT* to decide the satisfiability of the given formulae, i.e., they create a propositional formula, solve it with a SAT solver and propagate the results back to the higher-level theories.

The performance of these solvers is compared to two solvers being used in the EmbeddedPlatform: CBMC [13, 29] and iSAT3 [26]. Both solvers use their own input format for SMT formulae. CBMC itself is invoked by a wrapper that extends it with k-Induction. The resulting tool is referred to as EP-CBMC in the following. This wrapper also sets the option `--refine`. With this option, CBMC creates a weakened encoding of floating-point values (i.e., the number of models may increase during this step). This encoding is then refined step-by-step as long as models are found that do not fulfill the original floating-point formula. All other solvers are invoked without further command-line options that have an influence on the solving process; the exact commands may be found in Appendix B. While CBMC uses similar approaches as the SMT-LIB solvers, iSAT3 makes use of other ideas, namely Craig Interpolation [21] to prove the unsatisfiability of formulae and interval-based reasoning [27] instead of bitblasting. More details about the used solvers and their approaches may be found in [22].

All experiments for this thesis have been performed on a machine with 2 AMD EPYC® 7542 32-Core CPUs running at 2.9 GHz. Each of them launched a Windows® 10 Virtual Machine[3] with 32 virtual cores and 508 GB RAM, executing 32 separate solver processes in parallel. Most evaluations are based on a 60s time limit, as this value is used in several use-cases of the Embedded*Platform*®. Additionally, a step limit of 50 for BMC / k-Induction / Craig Interpolation has been applied.

Neither CBMC nor iSAT3 are using detection of constants to speed-up the solving process. Therefore, a good starting point is to compare them to the SMT-LIB solvers without any elimination of constants.

Figure 2 presents how many instances could be solved (i.e., a violation has been found, the unreachability has been proven or the maximum step bound of 50 has been reached) in a given amount of time. For example, after 48s, MathSAT5 has solved 5624 instances. It can be seen that CBMC and iSAT3 show a good performance on the given benchmark set and solve 8556 resp. 8069 instances. MathSAT5 does at least solve 5979 instances. The slowest solvers are Z3 and CVC4

[3] Windows has been used because the Embedded*Platform*® and all backend tools are only compiled for Windows® at the moment.

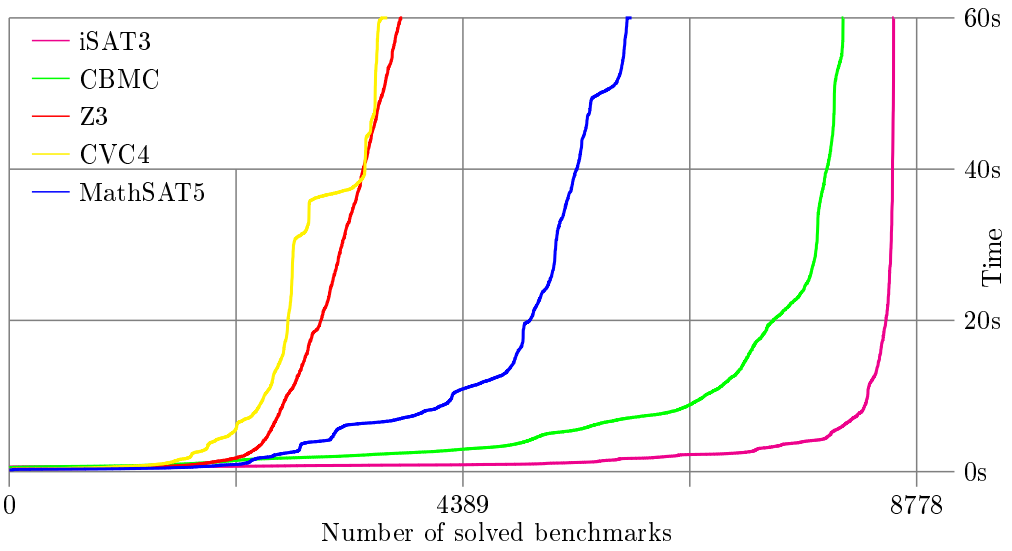

Figure 2: Performance with elimination of constants disabled, timeout 60s, averaged over 3 runs

with 3768 and 3592 instances.

The lead of iSAT3 in comparison to CBMC is mainly caused by the number of solved TRUE (i.e., unreachable) instances: While CBMC solves 631 of them, iSAT3 succeds on 970 instances, three to four times as many as the SMT-LIB solvers. This is due to Craig Interpolation in iSAT3 that performs better than k-Induction on finding TRUE results. When considering the number of detected FALSE results, CBMC even has a slight lead (7420 solved instances) in comparison to iSAT3 (7401 solved instances). The SMT-LIB solvers follow with 5577 (MathSAT5), 3510 (Z3), and 3334 (CVC4) solved benchmarks. The remaining solved instances have been unrolled until depth 50; neither BMC nor k-Induction could succeed until this depth. Still, the benchmark is considered as solved here.

Despite the number of solved instances, the depth where a TRUE / FALSE result can be gained is of interest during this thesis: If elimination strengthens the formulae so that solving can be finished at an earlier depth, the performance of solvers may improve noticeably, even if that is not the case for the solvers used for this evaluation.

Table 3 displays the number of benchmarks for which BMC returns a model at a given depth. This number should be consistent over all solvers, because a solver should find a model iff all other solver do so. Note that this data is collected from a file that has been created with the results of several BTC Embedded*Platform*® backend solvers (including CBMC and iSAT3). Because of that, the total number of FALSE benchmarks in this table is larger than the number of FALSE benchmark any SMT-LIB solver can process. It can be seen that for most of the FALSE benchmarks, a violating execution trace can be found in 5 steps. Note that all evaluations during this bachelor thesis are executed with a depth limit of 50, i.e., all instances with larger depths (114 instances) will lead to a MAXSTEP result if the solver reaches depth 50.

Accordingly, Table 4 presents the number of TRUE results gained at a particular depth. This analysis cannot use the results of other solvers, as iSAT3 uses Craig Interpolation that may need another number of depths to succeed (in fact, iSAT3 needs at least 4 Craig Interpolation steps for most of the benchmarks). Because MathSAT5 solves the most TRUE instances of all SMT-LIB

Depth	Number of benchmarks
1	2800
2	2612
3	1297
4	673
5	184
6	5
9	1
10	1
100	1
101	5
102	37
103	14
459	2
481	23
482	24
483	8
total	7687

Table 3: The number of FALSE benchmarks per depth.

Depth	Number of benchmarks
1	39
2	318
3	1
total	358

Table 4: The number of TRUE benchmarks per depth.

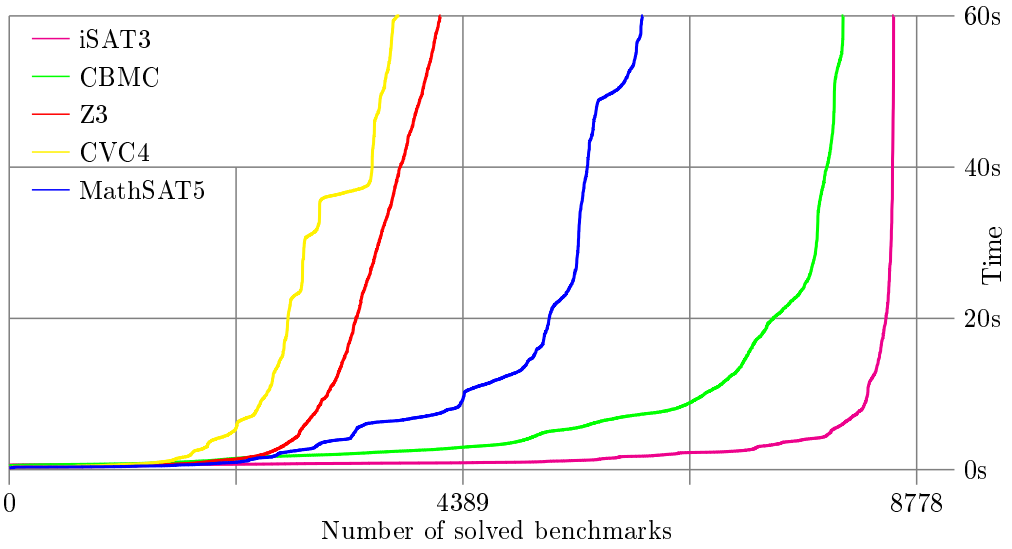

Figure 3: Performance with syntactic elimination of constants, timeout 60s, averaged over 3 runs

solvers, the needed depths are displayed here (this should be consistent at least for all SMT-LIB solvers). Obviously, the number of depths needed to prove the correctness of properties is very small: There is only one instance that requires 3-induction to be solved. When analyzing the CBMC results, a similar conclusion can be drawn. This is the reason why in multiple use-cases of the Embedded*Platform*®, only 2-induction is performed (which also catches instances that only need 1-induction).

In preparation for this thesis, a syntactic elimination approach has been developed to improve the SMT-LIB encoding. This approach identifies statements in the transition relations where a state variable gets assigned its value from the previous step, i.e., `x' = x`.

When enabling this syntactic elimination approach, the picture changes only slightly (Figure 3): As expected, the number of solved TRUE instances rises noticeably, while the number of solved FALSE instances remains almost constant. Only Z3 is able to solve significantly more FALSE instances using the syntactic elimination. It should be noted here that the amount of time needed for syntactic elimination is very small.

All of the additionally solved TRUE instances can be solved in depth 2; without elimination, none of them could be solved within 1 hour. Because of the weaker induction formulae, the reached induction depths (from 7 to 16) was not enough to show that the property could never be violated. The target of this thesis is to improve the performance further, i.e., to increase the number of solved instances or at least decrease the induction depth that the SMT-LIB solvers need to find a solution. To investigate whether this target has been reached, the performance of the several elimination modes must be evaluated.

4.3 Single Elimination

In Section 3.2, a technique has been described to check whether a single variable keeps its value. This approach is evaluated in this section. As a starting point, the elimination has been run with a

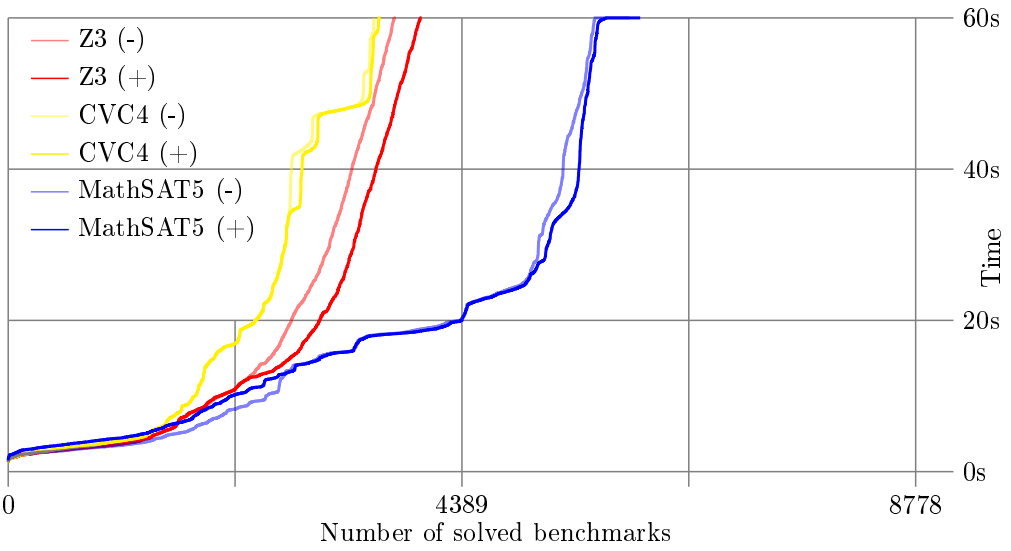

Figure 4: Performance with single elimination of constants, without (-) and with (+) syntactic elimination, timeout 60s, averaged over 4 runs

time limit of 20%[4] The number of iterations has not been limited, i.e., the elimination was repeated until either a fixpoint was found or the time limit has been reached.

Figure 4 shows the performance of the SMT-LIB solvers with these settings. When applying single elimination of constants, the number of solved FALSE instances decreases because single elimination consumes a significant amount of time that is not available for BMC / k-Induction solving anymore. On the other hand, the number of solved TRUE instances increases slightly, at least for Z3 and MathSAT5, in comparison to a run without any elimination. Furthermore, it is clear that syntactic elimination of constants outperforms the single variable approach in both categories: It is able to speed up k-Induction by identifying constants, but works in a minimal amount of time.

An important problem with the single elimination approach is that the time required for a check of each variable is not limited. Because of that, the number of variables being checked may be very small. Experimental data shows that this is indeed the case: CVC4 can only complete elimination for 3169 instances, while Z3 can finish it for 3306 and MathSAT5 for 3378 instances.

The depths needed to gain a result on the TRUE benchmarks are presented in Table 5. Apparently, many benchmarks that needed 2 steps are now solved within 1 step. From these results, it can be concluded that elimination of constants may indeed strengthen k-Induction formulae so that the solvers need less depths to prove a goal as unreachable.

When combining both the syntactic and the single variable approach, the performance changes as follows: All three SMT-LIB solvers solve less FALSE instances than with syntactic elimination of constants, but for Z3 and MathSAT5, the number of solved TRUE instances increases. In comparison to a run without any elimination of constants, the number of solved TRUE instances rises, while CVC4 and MathSAT5 gain less FALSE instances (for Z3, this number increases).

[4] Technically, the time for the translation to the SMT2 format is integrated into this limit. For example, a time limit of 20% with a total timeout of 60s means that this preparation is performed and after that, elimination is executed until the total execution time exceeds 12s.

Depth	Number of benchmarks
1	133
2	256
3	1
total	390

Table 5: The number of TRUE benchmarks per depth when applying single elimination of constants.

Depth	Number of benchmarks
1	183
2	256
3	1
total	440

Table 6: The number of TRUE benchmarks per depth when applying syntactic and single elimination of constants.

When looking at the number of depths that k-Induction needs, the results are more promising, as shown in Table 6: The number of instances that could be solved within one induction step increases further in comparison to using single elimination alone.

In conclusion, it has been shown that single elimination of constants can speed up k-Induction, while slowing down all other instances (an overhead that will also be present in all following approaches). At least, the depths necessary for k-Induction to succeed can be reduced by this approach.

4.4 Disjunctive Elimination

This section deals with the conjunctive elimination approach as described in Section 3.3. Again, the evaluations are performed with a time limit of 20%.

Figure 5 contains the performance of the several solvers when applying disjunctive elimination of constants. In comparison to single elimination, the number of instances solved by Z3 decreases; while MathSAT5 succeds on more TRUE instances. CVC4 is able to solve nearly the same number of instances as with single elimination. Except of MathSAT5, which solves more TRUE instances than without any elimination, the number of solved instances decreases in comparison to runs without or with only syntactic elimination enabled. This is caused by the fact that disjunctive elimination needs to finish in order to identify constants (the formula must be unsatisfiable). If a timeout occurs, the time needed for elimination is lost for BMC / k-Induction solving. This is indeed a very important problem: CVC4 can finish elimination on only 1982 instances. Z3 succeeds on 2790 and MathSAT5 on 3170 instances.

The picture changes when considering the depths in which k-Induction gains a result: Many benchmarks that needed depth 2 before can now be solved in depth 1. It seems that disjunctive elimination can strengthen the k-Induction formulae. This can potentially lead to a faster solver execution because each k-Induction step needs a significant amount of time.

When complementing the disjunctive approach with syntactic elimination, the performance

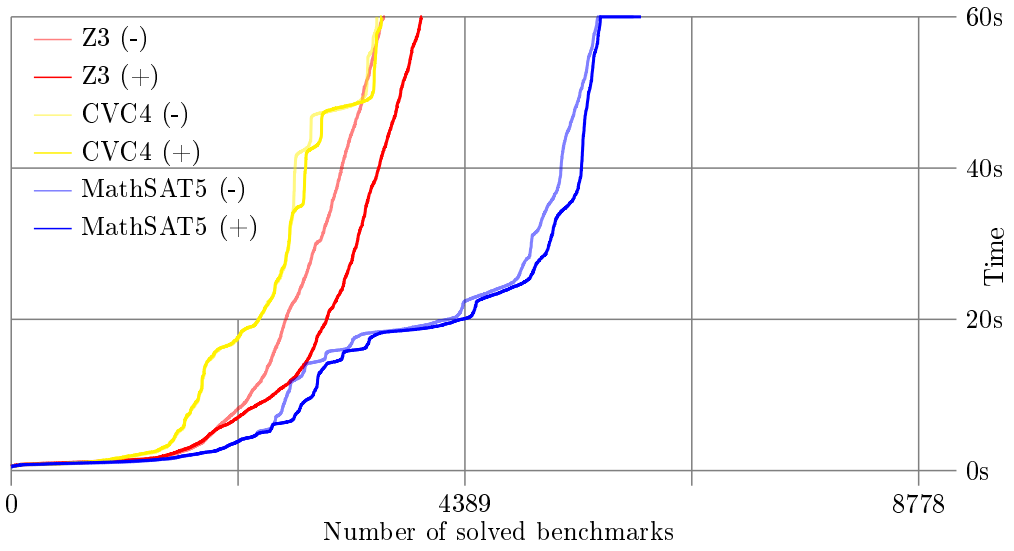

Figure 5: Performance with disjunctive elimination of constants, without (-) and with (+) syntactic elimination, timeout 60s, averaged over 3 runs

Depth	Number of benchmarks
1	253
2	153
3	1
total	407

Table 7: The number of TRUE benchmarks per depth when applying disjunctive elimination of constants.

Depth	Number of benchmarks
1	256
2	184
3	1
total	441

Table 8: The number of TRUE benchmarks per depth when applying syntactic and disjunctive elimination of constants.

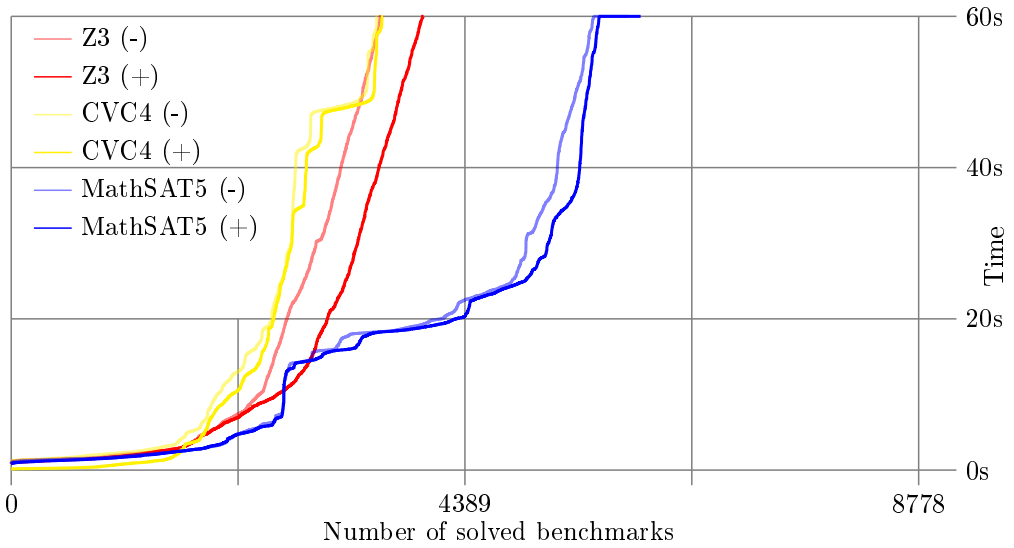

Figure 6: Performance with conjunctive elimination of constants, without (-) and with (+) syntactic elimination, timeout 60s, averaged over 3 runs

of all solvers increases. MathSAT5 can now also handle more FALSE instances than without any elimination. The two other solvers can at least solve more TRUE instances compared to this setting. However, no solver can outperform the performance gained on the combination of syntactic and single elimination of constants (at least, CVC4 solves nearly the same number of instances).

Disjunctive elimination seems to detect many constants that syntactic elimination can also find: The depth required for k-Induction does not decrease in comparison to a run with only disjunctive elimination enabled. The performance increase results from the faster execution of syntactic elimination.

In short, the disjunctive elimination approach can not increase the number of solved instances, but has a strong impact on the depths where a k-Induction result can be gained.

4.5 Conjunctive Elimination

The third approach described in this thesis is conjunctive elimination of constants (Section 3.4). It is evaluated here with the same settings regarding timeout as the other approaches.

The number of instances solved with conjunctive elimination is presented in Figure 6. Z3 and

Depth	Number of benchmarks
1	39
2	318
3	1
total	358

Table 9: The number of TRUE benchmarks per depth when applying conjunctive elimination of constants.

Depth	Number of benchmarks
1	41
2	380
3	1
total	422

Table 10: The number of TRUE benchmarks per depth when applying syntactic and conjunctive elimination of constants.

MathSAT5 solve even less instances than with disjunctive elimination, while CVC4 stays on the same level. In comparison to the syntactic elimination, the number of solved instances decreases for all solvers. An important reason is that the conjunctive elimination starts with the set of all constant candidates and splits it until the resulting formula is satisfiable or a single variable is reached. In the worst case, the solver needs $\log(n)$ steps (with n being the initial number of constant candidates) to get a definite result for the first variable. CVC4 was not able to complete the check of a single variable for 5423 instances; MathSAT5 and Z3 did not gain any information for 2351 resp. 4386 instances.

Conjunctive elimination of constants is not able to decrease the necessary induction depths, as shown in Table 9: The number of benchmarks that can be solved in a given depth is almost equal to a run without any elimination. This is caused by the amount of time needed for this approach, that is significantly larger than for disjunctive elimination: CVC4 needs 11.94s instead of 9.58s; Z3 9.02s instead of 8.70s. Only MathSAT5 can improve the time needed from 9.19s to 8.82s.

Again, when combining this approach with syntactic elimination, the performance increases. All three solvers succeed on a similar number of instances than with syntactic and disjunctive elimination, with the exception of MathSAT5 that loses some TRUE instances.

In short, this approach reaches essentially the same performance as disjunctive elimination on the given benchmark set, but does not do a good job in reducing the required induction depth.

4.6 Conjunctive Elimination using Strongly Connected Components

Section 3.4 not only introduces the general idea of conjunctive elimination, but also a more sophisticated approach to identify the initial subsets of variables that makes use of Strongly Connected Components.

When executing SCC detection on the normal dependency graph, the number of TRUE and FALSE instances solved by Z3 increases. MathSAT5 can solve less FALSE but more TRUE instances, while CVC4 stays at the same level in comparison to a run with normal conjunctive

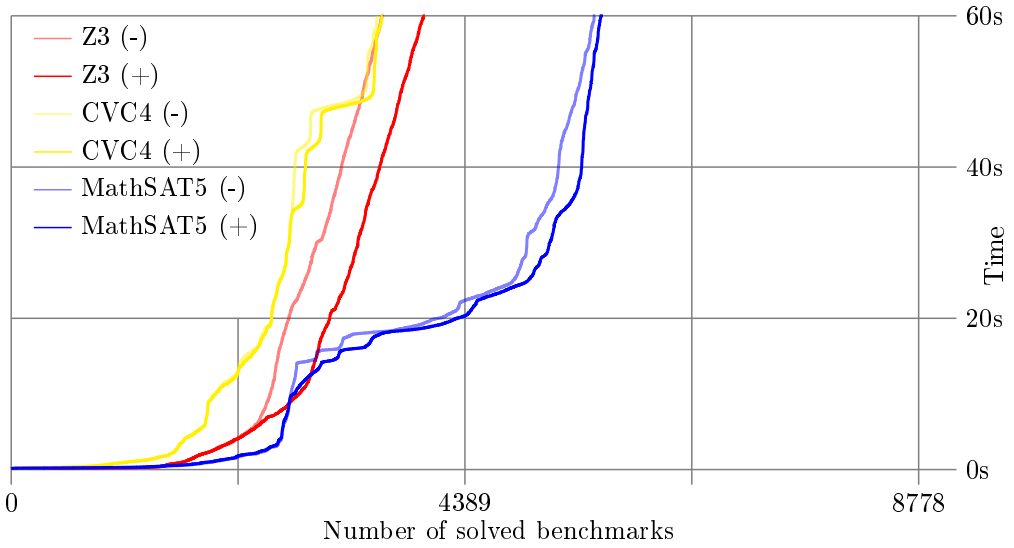

Figure 7: Performance when using the normal dependency graph, without (-) and with (+) syntactic elimination, timeout 60s, averaged over 3 runs

Depth	Number of benchmarks
1	39
2	326
3	1
total	366

Table 11: The number of TRUE benchmarks per depth when using the normal dependency graph.

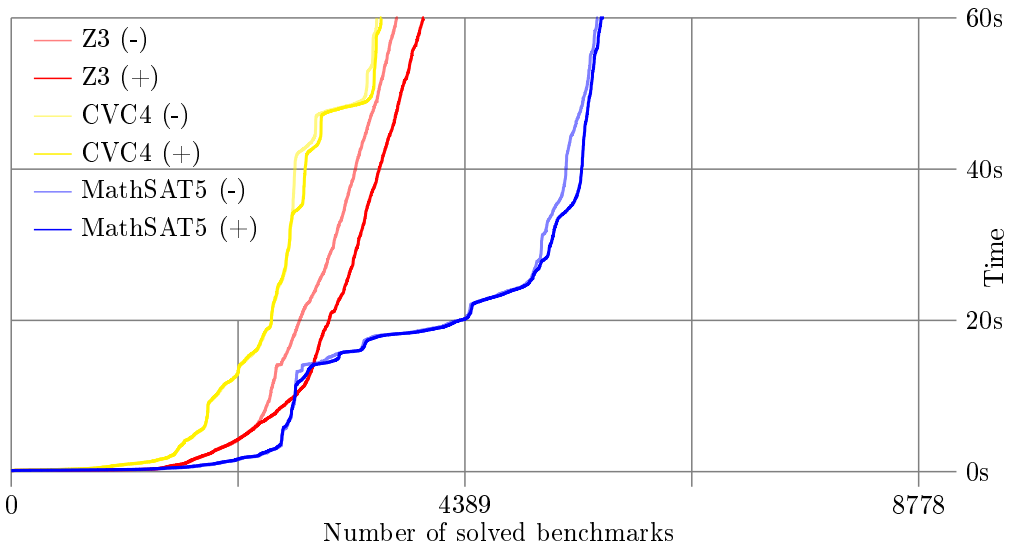

Figure 8: Performance when using the inverse dependency graph, without (-) and with (+) syntactic elimination, timeout 60s, averaged over 3 runs

Depth	Number of benchmarks
1	39
2	349
3	1
total	389

Table 12: The number of TRUE benchmarks per depth when using the inverse dependency graph.

elimination.

When using the inverse dependency graph, especially Z3 can improve its performance: It now solves 134 additional FALSE and 9 additional TRUE instances and can even outperform the disjunctive elimination. CVC4 can not improve with this setting, but MathSAT5 can also solve 7 more FALSE and 24 more TRUE instances than with the normal dependency graph. It should be noted that conjunctive elimination takes almost the same time in both configurations. The performance improvements result from the larger number of constants being found: While Z3 can identify 7289 constants in a run with SCCs from the normal dependency graph, this number increases to 8077 constants when using the inverse dependency graph.

It can be concluded that finding Strongly Connected Components in the inverse dependency graph can help to improve the performance of conjunctive elimination.

4.7 Other Elimination Timeouts for TRUE Instances

All previous evaluations have been performed with an elimination timeout of 20%. The target of this section is to evaluate other timeouts to find out whether this was a good choice. This evaluation is done on the known TRUE benchmarks only, because the target of this thesis is to

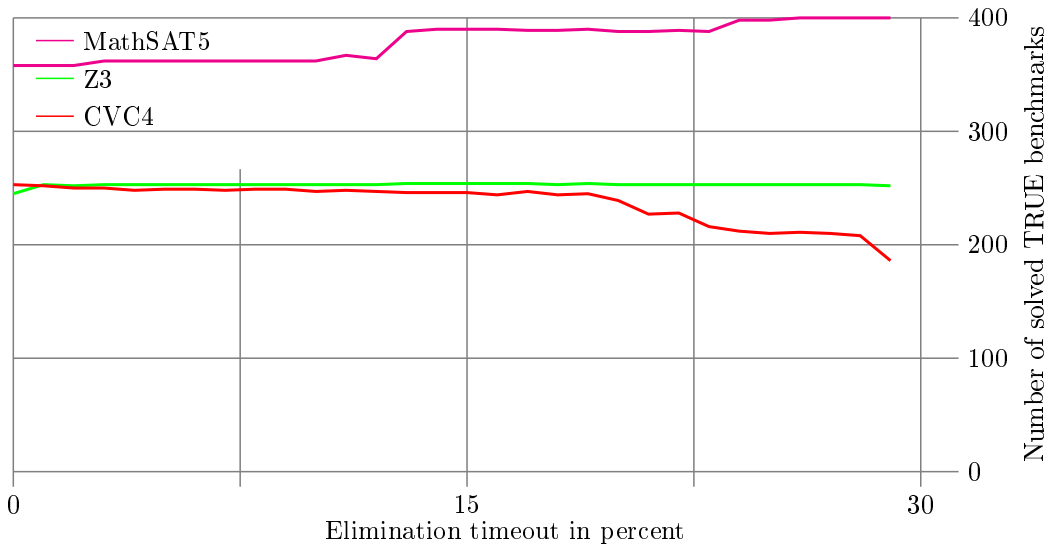

Figure 9: Solved TRUE instances with given elimination timeouts

speedup k-Induction. Furthermore, only the single elimination is considered here.

Figure 9 displays the number of TRUE benchmarks that have been finished by each solver using the given elimination timeout. CVC4 shows its best performance when elimination is disabled completely. MathSAT5 solves more TRUE instances when the elimination timeout is raised to 28%. Z3 almost solves the same number of instances when an elimination timeout of 2% is used.

The most FALSE instances are solved if only syntactic elimination is disabled. Especially for MathSAT5, the number of solved TRUE instances can be raised by increasing the timeout for elimination, which implies that less FALSE instances are solved, and vice versa. When using a timeout of 30%, more constants are detected. As a result, TRUE instances that could not be solved within 3 resp. 8 steps can now be solved in 2 steps. However, the depth of already solved instances could not be improved further.

To summarize, the optimal value for the elimination timeout is solver-specific: While Z3 is not affected by the chosen timeout and the performance of CVC4 even decreases, MathSAT5 can solve additional TRUE instances using larger elimination timeouts.

4.8 Improved Timeouts for Elimination

A serious problem with all elimination approaches is an occurring timeout: The current check is aborted without contributing any improvement to the encoded benchmark.

While the execution time of a single check cannot be predicted, it can be seen from the experimental results that in some cases, later iteration steps take more time than the previous ones, especially for disjunctive elimination. As an improvement, the *smart timeout* is introduced to the disjunctive and conjuctive elimination: If the last check took more time than is left now for elimination, the elimination is aborted. For single elimination, the elimination is limited to one iteration, i.e., all variables are only checked once.

For single elimination, the number of solved instances does not change significantly when limiting the number of iterations to 1. When combining the smart timeout with conjunctive elimination, the number of solved FALSE instances increases slightly for Z3 and decreases for MathSAT5. With

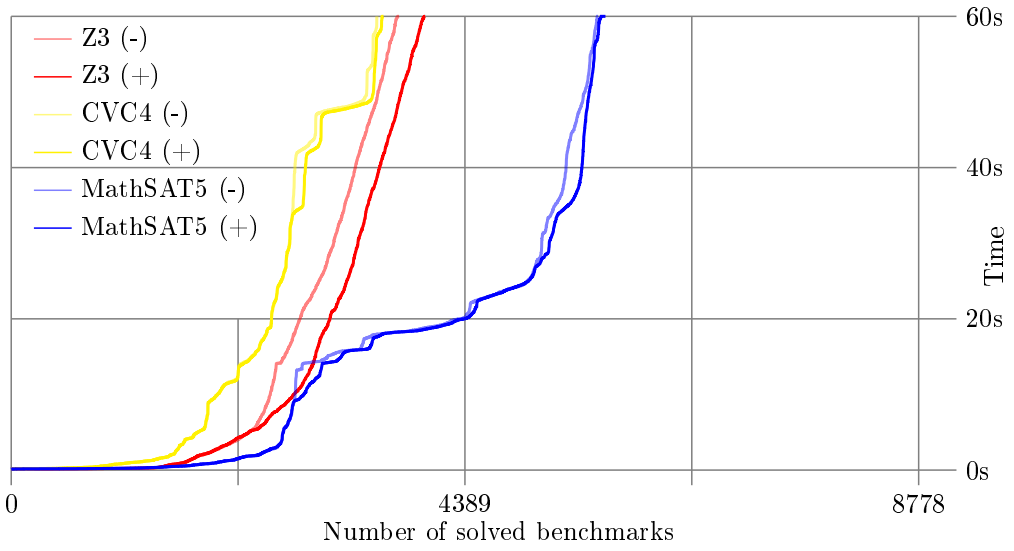

Figure 10: Single elimination with only one iteration, without (-) and with (+) syntactic elimination, timeout 60s

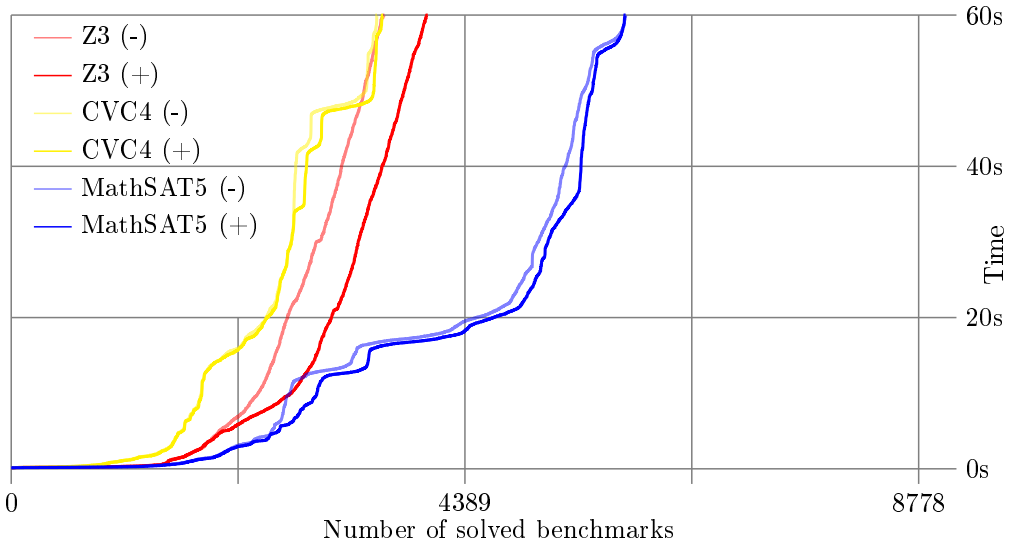

Figure 11: Disjunctive elimination, without (-) and with (+) syntactic elimination, smart timeout of 60s

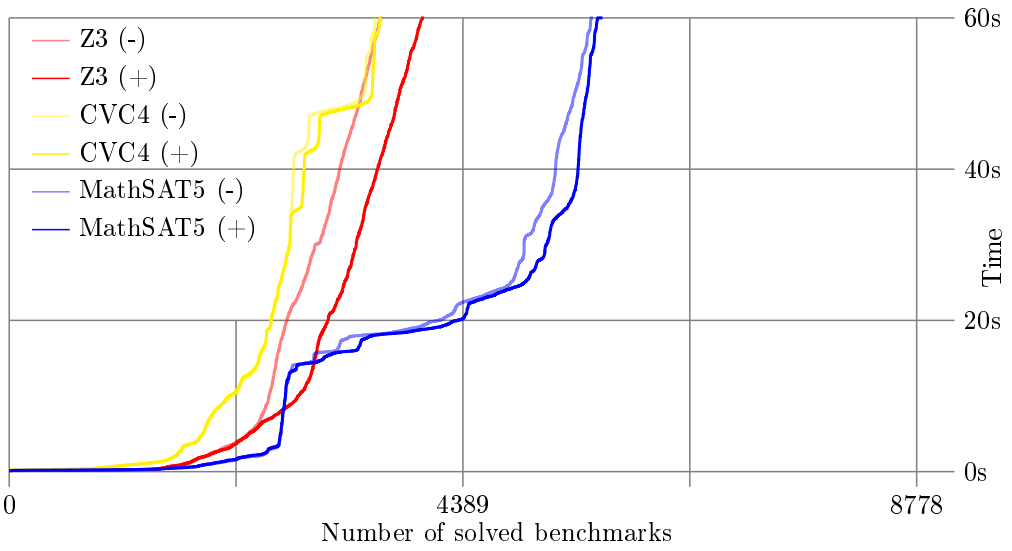

Figure 12: Conjunctive elimination, smart timeout of 60s

disjunctive elimination, the improvements are more noticeable, at least for MathSAT5: This solver now succeeds on 270 more FALSE instances than before. The reason is that this strategy terminates very complex elimination tasks earlier, resulting in a decrease of average time needed from 9.22s to 7.22s, without loosing much information about constants (the number of solved TRUE instances decreases from 404 to 399).

4.9 Performing 2-induction Only

During the evaluation, it has been shown that k-Induction is successful in depth 1 or 2 for the most TRUE instances. Because of that, some use-cases in the Embedded*Platform*® only perform k-Induction in depth 2 by default. While decreasing the time needed for BMC and all induction steps with depth 2, instances that can be solved in depth 1 will take longer, because the 2-induction check does in general consume more time than 1-induction. Instances that require at least three induction steps cannot be solved anymore.

When performing an induction check in depth 2 only, the number of solved instances increases for almost all settings and solvers. In general, Z3 can benefit most from this; it solves a lot more instances in all settings. For example, the number of FALSE instances solved with disjunctive elimination enabled raises from 3340 to 3832 and the number of TRUE instances from 246 to 416.

Without the semantic elimination strategies described in this thesis, MathSAT5 and CVC4 solve less instances than with full induction enabled. In combination with any such strategy, they can improve their performance if only 2-induction is performed.

It can be concluded that executing 2-induction only can help the solvers, especially Z3, to solve more instances within the given time limit. For the other solvers, it may be an option to run both 1- and 2- induction to improve their performance further.

Z3 performance summary				
Configuration	false	true	maxstep	timeout
No elimination	3510	245	13	5009
Syntactic elimination	**3799**	366	13	**4509**
Single elimination	3471	254	13	5039
Syntactic + single elimination	3603	374	13	4787
Disjunctive elimination	3340	264	13	5178
Syntactic + disjunctive elimination	3585	**381**	13	4798
Conjunctive elimination	3321	244	13	5190
Syntactic + conjunctive elimination	3595	374	13	4794
Conjunctive elimination, SCC, normal dep. graph	3333	245	13	5178
Conjunctive elimination, SCC, inverse dep. graph	3467	254	13	5042
Single elimination, one iteration only	3473	254	13	5038
Disjunctive elimination, smart timeout	3348	244	13	5172
Conjunctive elimination, smart timeout	3338	243	13	5183

Table 13: Performance of Z3 over all configurations

4.10 Summary

The results of this section are summarized in one table per solver that mentions the number of solved FALSE / TRUE / MAXSTEP benchmarks as well as the number of timeouts per configuration.

5 Conclusion and Future Work

5.1 Conclusion

The target of this thesis was to develop and evaluate approaches to identify constants in given BMC / k-Induction problems to speedup the necessary checks. After describing the fundamental theoretical concepts, three approaches have been introduced: single, disjunctive and conjunctive elimination of constants. They all use a SMT-LIB solver to identify constants, but introduce different strategies regarding the structure of the formulae. During the evaluation, these three approaches (together with some promising variants) have been applied to a benchmark set.

The evaluation has shown that elimination of constants can improve the solver performance in two ways: On the one hand, the solvers are able to succeed on more instances within the same amount of time. On the other hand, elimination reduces the depth that k-Induction needs to prove the validity of the given property. The largest reduction of the required k-Induction depth is gained by the disjunctive elimination, followed by the single elimination strategy. The conjunctive elimination takes the last place in this comparison, although its performance can be improved by a more intelligent selection of variable subsets, for example using Strongly Connected Components.

While detection of constants can increase the number of solved TRUE benchmarks significantly, it consumes runtime that is missing for BMC and k-Induction checks. Syntactic elimination can also speedup the solving process for FALSE instances, but the elimination approaches presented in this thesis consume so much time that they have a negative impact on the FALSE instances.

During this thesis, the performance of the three solvers Z3, CVC4, and MathSAT5 has been evaluated. This three solvers show different changes in the number of solved instances when elimination of constant is applied: Z3 works best with the disjunctive elimination strategy, together

CVC4 performance summary				
Configuration	false	true	maxstep	timeout
No elimination	3334	253	**5**	5185
Syntactic elimination	**3444**	**301**	**5**	**5027**
Single elimination	3291	245	1	5240
Syntactic + single elimination	3306	280	1	5190
Disjunctive elimination	3291	244	1	5242
Syntactic + disjunctive elimination	3306	280	1	5191
Conjunctive elimination	3290	244	1	5242
Syntactic + conjunctive elimination	3308	281	1	5187
Conjunctive elimination, SCC, normal dep. graph	3307	280	1	5189
Conjunctive elimination, SCC, inverse dep. graph	3303	277	1	5196
Single elimination, one iteration only	3291	245	1	5241
Disjunctive elimination, smart timeout	3292	244	1	5241
Conjunctive elimination, smart timeout	3292	245	1	5239

Table 14: Performance of CVC4 over all configurations

MathSAT5 performance summary				
Configuration	false	true	maxstep	timeout
No elimination	5577	358	44	2798
Syntactic elimination	**5560**	422	43	**2652**
Single elimination	5275	390	43	3069
Syntactic + single elimination	5306	**440**	43	2989
Disjunctive elimination	5234	404	43	3096
Syntactic + disjunctive elimination	5238	441	43	3055
Conjunctive elimination	5245	357	43	3132
Syntactic + conjunctive elimination	5239	422	43	3073
Conjunctive elimination, SCC, normal dep. graph	5232	365	44	3137
Conjunctive elimination, SCC, inverse dep. graph	5239	389	44	3105
Single elimination, one iteration only	5238	389	44	3106
Disjunctive elimination, smart timeout	5504	399	44	2831
Conjunctive elimination, smart timeout	5231	357	44	3145

Table 15: Performance of MathSAT5 over all configurations

with a small timeout (ca. 5 percent). For CVC4, disabling all elimination strategies (except syntactic elimination) completely seems to be the best choice. MathSAT5 shows the best performance for TRUE instances when an elimination timeout of 30% is applied to the disjunctive elimination strategy. On the other hand, the most number of FALSE instances can be solved with a timeout of 10%. No solved is able to reach the same performance as CBMC or iSAT3 on the given benchmark set. The next section lists some ideas that may help to reduce the gap further.

5.2 Future Work

The ideas proposed in this thesis have the potential to speedup k-Induction solving significantly. Nevertheless, there is still room for improvements that can lead to decreased runtimes and k-Induction depths.

Section 3.1 explained why BMC is slowing down when being combined with k-Induction. To circumvent this problem, the original article that presented k-Induction [16] recommended to use two separate solver instances: One of them performs all BMC steps, while the other one solves the k-Induction formulae. This approach can even be improved further by running the two instances in parallel. The tool smi_smt2 has been extended recently by such a parallelization mode. On the one hand, elimination of constants could be parallelized itself by running two solver instances, each checking one half of the constant candidates. On the other hand, elimination can be performed on the k-Induction instance only; the BMC instance starts solving directly. This would avoid the decrease in the number of solved FALSE instances that has been observed.

This thesis also shows that simple syntactic elimination is very efficient: It needs very little time, but can still detect several constants. Therefore, if this syntactic elimination can be improved by extending it with more cases, elimination can be improved further. To identify such cases, the approaches described in this thesis can be used.

Another potential for optimization lays in a careful selection of the variables being solved together in the disjunctive / conjunctive elimination. The strongly connected components are a first step in this direction, but other heuristics may still improve the performance.

This thesis has only considered "model-complete" SMT-LIB solvers: If a model exists, these solvers find it. This is not the case for solvers such as XSAT [17] and GoSAT [8]: They use mathematical optimization to guess models for the given formulae. If these tools return SAT, the resulting model should be correct, but if they do not find a model, this does not prove the unsatisfiability of the given property. When running these solvers in parallel with a complete solver, the performance of the checks with satisfiable formulae may improve significantly.

To summarize, there is still room for improvements that can make elimination of constants even more applicable for improving the performance of k-Induction.

References

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

[2] Haniel Barbosa, Jochen Hoenicke, and Antti Hyvarinen. *The 15th International Satisfiability Modulo Theories Competition (SMT-COMP 2020).* URL: `SMT-COMP.github.io/2020/`.

[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 Standard: Version 2.6.* Tech. rep. Department of Computer Science, The University of Iowa, 2017.

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

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_14.

[7] 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.

[8] 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, Vienna, Austria, October 2-6, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, pp. 11–14. DOI: 10.23919/FMCAD.2017.8102235. URL: https://doi.org/10.23919/FMCAD.2017.8102235.

[9] 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.

[10] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819. URL: https://doi.org/10.1109/TC.1986.1676819.

[11] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 93–107. DOI: 10.1007/978-3-642-36742-7_7. URL: https://doi.org/10.1007/978-3-642-36742-7%5C_7.

[12] 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.

[13] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15. URL: https://doi.org/10.1007/978-3-540-24730-2%5C_15.

[14] 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.

[15] 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.

[16] 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.

[17] Zhoulai Fu and Zhendong Su. "XSat: A Fast Floating-Point Satisfiability Solver". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 187–209. DOI: 10.1007/978-3-319-41540-6_11. URL: https://doi.org/10.1007/978-3-319-41540-6%5C_11.

[18] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

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

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

[21] Kenneth L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1. URL: https://doi.org/10.1007/978-3-540-45069-6%5C_1.

[22] 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.

[23] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (1990), pp. 32–44. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[25] Silvio Ranise, Cesare Tinelli, and Clark Barrett. *The SMT-LIB quantifier-free bit vector logic*. URL: www.SMT-LIB.org/logics-all.shtml#QF_BV.

[26] 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.

[27] Karsten Scheibler, Felix Neubauer, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, Detlef Fehrer, and Bernd Becker. "Accurate ICP-based floating-point reasoning". In: *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. Ed. by Ruzica Piskac and Muralidhar Talupur. IEEE, 2016, pp. 177–184. DOI: 10.1109/FMCAD.2016.7886677. URL: https://doi.org/10.1109/FMCAD.2016.7886677.

[28] Uwe Schöning. *Logik für Informatiker, 5. Auflage*. Spektrum-Hochschultaschenbuch. Spektrum Akadem. Verl., 2000. ISBN: 978-3-8274-1005-4.

[29] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. "Incremental bounded model checking for embedded software". In: *Formal Aspects Comput.* 29.5 (2017), pp. 911–931. DOI: 10.1007/s00165-017-0419-1. URL: https://doi.org/10.1007/s00165-017-0419-1.

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

[31] 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.

[32] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms (Working Paper)". In: *12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971*. IEEE Computer Society, 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10. URL: https://doi.org/10.1109/SWAT.1971.10.

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

[34] Cesare Tinelli and Martin Brain. *The SMT-LIB FP theory*. URL: www.SMT-LIB.org/theories-floatingpoint.shtml.

A Files Generated by smi2engine

As mentioned in Section 2.5, the tool smi2engine is used to translate the SMI files into into five different SMT-LIB files.

For the program in Example 2.13, the generated SMT-LIB instructions are shown in Example 2.17. The complete files (including comments / additional variables) may look like this:

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)
```

B Solver Commands Used for the Evaluation

The following commands are used to invoke the several SMTLIB solvers:

```
z3 --in
cvc4 --incremental --lang=smt2.6 --output-lang=smt2.6
mathsat -printer.bv_number_format=2
```