University of Passau
Faculty for Computer Science and Mathematics

Master's Thesis

Light-Weight Invariant Generation for Software Verification with CPAchecker

Matthias Dangl

October 10, 2013

Master's Thesis
at the Software Systems Lab
at the Faculty for Computer Science and Mathematics
at the University of Passau

Assessor: Prof. Dr. Dirk Beyer
Advisor: M. Sc. Philipp Wendler

Abstract

This thesis presents a light-weight approach to invariant generation in the context of an implementation of k-induction for software verification within the CPAchecker framework. CPAchecker is a tool for software verification, one of the grand challenges of computing research. Inductive methods to prove the correctness of loops have been applied for years, but rely on manual invariant annotation or complex and elaborate algorithms for automatic generation of invariants, because classic inductive methods often require strong invariants to be able to prove loop correctness. k-induction, however, is often successful with weaker invariants.

The presented invariant generation algorithm supporting the k-induction implemented in the bounded model checking algorithm of CPAchecker is evaluated and it is shown that while k-induction certainly is no all-encompassing approach to software verification, there are promising results for a significant number of experiments, which might be an incentive to further research into the question what types of programs benefit most from a k-induction verification approach.

Contents

List of tables vii

List of Figures x

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Related Work 2
- 1.3 Terminology 6
  - 1.3.1 Programs and Control Flow Automata 6
  - 1.3.2 Types 6
  - 1.3.3 Pure Expressions 7
  - 1.3.4 SMT Solver 7

2 Background 9
- 2.1 k-Induction 9
- 2.2 Basics of Invariant Generation 10
  - 2.2.1 Information Extraction 15
  - 2.2.2 Information Representation 16
  - 2.2.3 Abstract Interpretation 16
- 2.3 Configurable Program Analysis 19

3 Theory 23
- 3.1 Providing k-Induction with Invariants 23
- 3.2 Invariant Generation 24
  - 3.2.1 Efficient Representation 24
    - 3.2.1.1 Sets of Integer Values 24
    - 3.2.1.2 Variable Environment 26
    - 3.2.1.3 Path Distinction 27
  - 3.2.2 Expression Evaluation and Abstraction 29
- 3.3 A Configurable Program Analysis for Generating Invariants 31

4 Implementation 33
- 4.1 Overview 33
- 4.2 InvariantsCPA 34
  - 4.2.1 Initialization 34
  - 4.2.2 Configuration 35

4.3 InvariantsDomain 35
4.4 SimpleInterval 38
4.5 CompoundState 38
4.6 InvariantsFormula 39
4.7 InvariantsState 40
4.7.1 Assignment Transitions 41
4.7.2 Assume Transitions 42
4.7.3 Joining States 43
4.7.4 Invariant Extraction 44
4.8 InvariantsTransferRelation 44
4.9 Evaluation of Formulas 45

5 Experimental Results 47
5.1 Selected Benchmarks 47
5.2 Analyses 47
5.2.1 Bounded Model Checking 47
5.2.2 Predicate Analysis 48
5.3 Measured Properties 48
5.3.1 Safety 48
5.3.2 Time and Space 49
5.4 Benchmark Environment 49
5.5 Experiments 49
5.5.1 Proof of Concept 49
5.5.2 Guessing of Potentially Interesting Variables 54
5.5.3 Comparative Benchmark 60
5.5.4 Related Benchmarks 63

6 Conclusion 65
6.1 Summary 65
6.2 Prospects 66
6.2.1 Integration into the existing framework 66
6.2.2 Multi-Loop Programs 66
6.2.3 Bit-Vectors 67
6.2.4 Pointer Aliasing 67
6.2.5 Further Heuristics 67

Bibliography 69

List of Tables

4.1 Configuration options . . . 36
4.2 Important external configuration options . . . 37

5.1 Configuration options used for the bounded model checking experiments on the `ssh` program selection . . . 51
5.2 Bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set . . . 52
5.3 Bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation . . . 55
5.4 CPU times and memory consumption for the different values of k when using bounded model checking with k-induction and invariant generation, setting `cpa.invariants.interestingVariableLimit` to `0` . . . 57
5.5 Bounded model checking with k-induction applied on the `ssh` benchmark set, using `cpa.invariants.interestingVariableLimit = 1` . . . 61
5.6 Comparison between three different analyses over the large benchmark set . . . 62

List of Figures

2.1 The control flow graph of a very simple program 11
2.2 The control flow graph of a very simple program with an error assertion 12
2.3 Another control flow graph of a very simple program with an error assertion 13
2.4 Another control flow graph of a very simple program with an error assertion 14
2.5 A control flow graph of a program containing a loop 17

3.1 A list of intervals representing $\top$ 26
3.2 An empty list of intervals representing $\bot$ 26
3.3 A list of intervals representing all integer numbers but zero 26
3.4 A list of intervals with two finite intervals 27
3.5 A complex list of intervals 27
3.6 An example control flow graph of a program where variable ranking might be sub-optimal 30

4.1 A mapping of variables to expressions, representing knowledge about an environment 41

5.1 The verification result chart for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set. In total there are 22 safe and 23 unsafe programs in the set. 51
5.2 The consumed CPU time graph for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set 53
5.3 The consumed memory graph for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set 53
5.4 The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation. In total the set contains 22 safe and 23 unsafe programs. 54
5.5 The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation 56
5.6 The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation 56

5.7 The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing one important variables. In total the set contains 22 safe and 23 unsafe programs. 57
5.8 The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing important variables 58
5.9 The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing important variables 58
5.10 The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable. In total the set contains 22 safe and 23 unsafe programs. 59
5.11 The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable 59
5.12 The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable 60

1 Introduction

1.1 Motivation

According to Tony Hoare, automatic verification of software is one of the grand challenges for computing research. He describes the correctness of computer programs as ”the fundamental concern of the theory of programming and of its application in large-scale software engineering” [1, p. 65] and emphasizes the importance of achieving the goal of automatic software verification not only for end-users but also for the economy as a whole [1, p. 66].

One of the criteria Hoare gives for a problem to be considered a grand challenge is that it must actually be a challenge [1, p. 64]. Verifying software is undeniably a challenging task, considering that even the question of whether or not a program terminates at all is undecidable in general[2].

The systematic process of using mathematical techniques to verify the conformity of a design specification to the design implementation is called formal verification[3]. An important aspect of formal software verification is the application of assertions over program variables, because whether or not an undesirable program state is reached usually depends on the values of those variables. Assertions over program variables that remain true each time a program location is reached are called invariant assertions[4].

One approach to formal software verification is model checking, which is described as an algorithmic analysis of programs used to prove their execution properties[5]. The basic idea of model checking is checking whether the finite state graph of a program is a model of its specification or not [6, p. 76].

This approach, however, is faced with the problem of representing large numbers of states for complex systems. To handle such situations, McMillan proposed the application of ordered binary decision diagrams (OBDDs) to represent boolean formulas in a compact canonical form. But the efficiency of these data structures depends on the NP-complete problem of optimal variable ordering for OBDDs and there are cases where even for an optimal ordering the size of the OBDD is proven to be exponential in the number of variables [6, p. 76].

Bounded model checking (BMC) tries to avoid this explosion of states by using propositional decision procedures instead of BDDs and considers counterexamples to the specification with a specific length k where the maximum length of the counterexample is limited by a bound[7].

To be able to prove correctness of loops without having to unroll them completely, which is often infeasible and sometimes impossible, inductive approaches are applied. For induction to be successful, sufficiently useful invariant assertions are required. Automatically generating useful invariants is, however, a hard task[8].

CPAchecker is a configurable software verification tool which implements, among other strategies, bounded model checking and provides a framework for implementing configurable program analyses[9]. The bounded model checking implementation of CPAchecker uses k-induction to prove loop correctness. While this advanced algorithm improves the ability to verify loop correctness, it still relies on sufficiently useful invariants, but contrary to traditional inductive approaches, k-induction is often successful with significantly weaker invariants[8]. Therefore, a light-weight approach to invariant generation has now been implemented as a configurable program analysis in CPAchecker, to be used by CPAchecker's bounded model checking algorithm.

1.2 Related Work

In 1967, Floyd described program verification methods using verification conditions in control flow graphs[10] and in the same year proposed a verifying compiler[11]. He was supported by King in 1970, who in his thesis presents a theoretical basis for program verifiers as a "first step toward developing a 'verifying compiler'"[12] and suggests using invariant statements as inductive predicates [12, p. 22 ff.] for software verification.

Two years later, Hoare elaborated on the basic techniques that can be used to explore the logical foundations of computer programming. Among the axioms and rules he provided to accomplish this task, he proposed the application of invariant assertions to the analysis of `while`-loops[13], which related to Floyd's suggestion of invariant statements as inductive predicates.

Wegbreit discussed heuristic approaches for synthesizing loop predicates extracted from partially manually specified inductive assertions and from loop boundary conditions as well as methods to extract loop predicates by using input predicates in combination with evaluation of weak interpretations [14, p. 102] in 1974. He concluded that the techniques he presented were not sufficient to provide all-encompassing invariant assertions for complex programs, but that an achievable goal might be to automatically provide a programmer with enough information to easily understand the more complex loops while covering the simpler loops without requiring interaction [14, p. 112].

Caplain attempted to extract the "Most General Invariant" [15, p. 165] of any program loop in 1975. He claimed that the challenging part of verifying program correctness is finding the inductive assertions required to prove the correctness of the loops contained in the programs [15, p. 165] and gave instructions to transform a loop into a partial specification of the loop containing a set of invariant assertions [15, p. 170].

Two years after Wegbreit's article about the synthesis of loop predicates, Mannar and Katz showed techniques for automatically deriving invariants and defined criteria required to use those generated invariants to prove correctness, incorrectness and termination [16, p. 188]. They pointed out that the general problem of finding invariants for any program is unsolvable and agree with Wegbreit on his suggestion that user interaction might always play a key role in software verification [16, p. 204].

Patrick Cousot and Radhia Cousot proposed abstract interpretation as a simple means to extract abstract information about the execution of a program without actually executing the program code exactly. They proposed using the resulting correct but inexact information to provide answers about partial correctness or similar questions, where more exact data is not always required [17, p. 238]. They postulated that most program analysis techniques may be seen as cases of abstract interpretation and point out several examples, one of them being the discovery of inductive invariants [17, p. 250].

In 1978, Cousot and Halbwachs explored methods to extract linear restraints among variables of programs[18]. They presented formal representations for their assertions and mapped their linear restraint transformer to basic elements of the language [18, p. 85].

Bryant suggested using binary decision diagrams with restrictions to variable ordering to represent boolean functions in 1986, thus introducing the concept of ordered binary decision diagrams (OBDDs). He presented algorithms for manipulating these decision diagrams and demonstrated his ideas on examples from logic design verification [19, p. 1]. While showing the efficiency of ordered binary decision diagrams for many cases, he also proved that there are functions that can never benefit from variable ordering in binary decision diagrams [19, p. 21 ff.].

In 1990, Burch et al. devised a method to symbolically represent state spaces of finite state systems by using the ordered binary decision diagrams described by Bryant and applied it to a model checking algorithm [20, p. 428]. They were able to reduce the ”complexity of various graph based verification algorithms” but point out that even though ordered binary decision diagrams were beneficent to their work, it might be desirable to adapt a representation structure not suffering from the drawbacks of OBDDs already mentioned by Bryant [20, p. 438].

This desired alternative to ordered binary decision diagrams was presented by Biere et al in 1999. They proposed bounded model checking, a technique were they reduced model checking to propositional satisfiability. They emphasized the benefits of SAT-based procedures for symbolic model checking and saw opportunities to further improve bounded model checking by combining it with other state-space reducing methods[7].

Hoare proposed a set of properties defining a grand programming challenge in 2003. He used software verification in form of the verifying compiler as his prime example for a grand programming challenge and elaborated on the various aspects of the problem making it so important and challenging to computing research. He proclaimed Floyds efforts in the late sixties and the seventies as failed due to insufficient availability of required fundamental research but was convinced that scientific progress since then might prove to enable more current efforts to succeed in this discipline[1].

Colón et al. in 2003 claimed to present ”the first sound and complete technique for generating inductive invariants” being complete for inductive invariants[4]. They drew the conclusion that their technique, while guaranteed to find all linear invariants of a linear program provable by inductive linear assertions, was not fit to be applied to larger systems due to the generation of non-linear constraints[4].

In early 2004, Sankaranarayanan et al. presented a method to generate non-linear invariants of a program. They reduced invariant generation to a numerical constant solving problem by applying the theory of ideals over polynomial rings. They mentioned that their approach of using constraint solving eliminates the impact of the polynomial degree of the desired invariants and stated that their technique would scale to larger applications. They did, however, admit that their method lacked completeness which might in some cases be undesirable[21].

Later that year, Rodríguez-Carbonell and Kapur showed a technique for generating polynomial loop invariants for imperative program code similar to the approach presented by Sankaranarayanan et al. Contrary to Sankaranarayanan et al., Rodríguez-Carbonell and Kapur provided an algorithm that is not only correct but also complete for the class of loops considered, and do not require the extensive usage of heuristics[22].

In 2006, Ernst et al. published their work about Daikon, a system they created to dynamically detect likely invariants of a program. Their dynamic technique is a machine learning approach to invariant detection. [23, p. 35] They emphasized that their tool is scalable, robust and extensible [23, p. 44].

Beyer et al. explained in 2007 that while they noted a theoretical convergence of program analysis and model checking, practical implementations had often remained ununifiable. Their solution to this problem was an algorithm for configurable program analysis, allowing both sides to co-operate. To prove their concept they implemented it in the software model checking tool Blast. Their experiments suggested that this approach also allowed customizations to software verification techniques leading ”to dramatic improvements in the precision-efficiency spectrum” [24, p. 504].

In 2009, Wahl gave a concise explanation of k-induction as a generalization of the standard induction which corresponds to 1-induction in the k-induction terminology. He proved not only that it is a valid method for constructing proofs but also that k-induction is superior to 1-induction[25].

Also in 2009, Clarke et al. composed a comprehensive scientific history of model checking. They provided a definition for formal verification and explained the basic idea of model checking, before discussing the state explosion problem and highlighting the advantages of model checking in general as well as the different approaches to model checking and their specific advantages and disadvantages[6].

In the same year, Gupta and Rybalchenko published their work about their software InvGen, a tool used to automatically generate linear arithmetic invariants for imperative programs. The techniques they use to generate the invariants are abstract interpretation and template generation[26].

In 2011, Beyer et al. presented their software verification tool CPAchecker based on the algorithm for configurable program analysis (CPA) they suggested in 2007[9][24]. Their goal was to provide an extensible platform to accelerate the process of transforming theoretical verification ideas into actual executable and testable software verfification programs [9, p. 184]. Their results showed that the predicate analysis they implemented in CPAchecker in most cases outperformed the model checking tool Blast. They also implemented an explicit-value analysis and integrated the C bounded model checker CBMC into CPAchecker, both producing experimental results they were positively impressed with [9, p. 189].

In the same year, Donaldson et al. claimed to be the first to apply k-induction to automatic loop verification [27, p. 1] using a technique they would later refer to as split-case k-induction[8]. They pointed out that by automatic translation of multi-loop programs into programs with only one single loop, they were able to cancel the negative impact of their technqiue's restriction to single-loop programs and demonstrated functionality and effectiveness of their idea by implementing it in a verification tool they named Scratch [27, p. 30].

Later that year, Donaldson et al. again employed k-induction to loop verification, proposing a technique they called combined-case k-induction for software verification. They now tried to handle multi-loop programs by cutting one loop at a time until the resulting program is loop free, thus eliminating the need of automatically transforming all program loops into a single one. Additionally they showed that for software verification to succeed with combined-case k-induction, the invariants used may often be significantly weaker than when using standard inductive invariant or split-case k-induction approaches[8].

To follow Donaldson et al.'s example, k-induction was recently integrated into the bounded model checking implementation of the software verification tool CPAchecker developed by Beyer et al. This thesis provides the invariant generation required to provide the information necessary to successfully apply the k-induction to loop verification in CPAchecker.

1.3 Terminology

1.3.1 Programs and Control Flow Automata

As the invariant generation algorithm presented in this thesis is implemented in the CPAchecker framework, which currently is used to analyze C code, the configurable program analysis generating the invariants was also tested on C code. The example programs shown in this thesis, however, are presented in the form of control flow graphs and are synonymously referenced as programs and control flow automata. Also, the control flow graphs shown will not include variable declarations; any previously unassigned variables appearing in the graph are simple treated as having an unknown value at that location. There are several reasons for this convention.

First and foremost, configurable program analyses like the implemented invariant generation algorithm operate on control flow automata in CPAchecker. The algorithm is thus not limited to C code; in theory it can be applied to any control flow automaton, although language specific code structures might introduce new kinds of edges that could require extra handling.

Secondly, control flow graphs make the state transitions of the programs more obvious. It is therefore easier for the reader to retrace which transitions are possible and which ones are not.

Thirdly, it is often desirable to take a closer look at an excerpt of a program, where not all information about the program is presented. This mirrors the frequent real situations, where a part of a program is supposed to be verified, but some information from external sources such as third-party libraries or user input is not known to the analysis. In practice, this missing information is often simulated by replacing occurrences with calls to functions returning random values. In this thesis, this would unnecessarily bloat the examples. Showing only a control flow graph that lacks such missing information is probably less confusing to a reader than them stumbling upon distracting calls to randomizers or having to remember how a specific language treats uninitialized variables.

1.3.2 Types

The implemented invariant generation algorithm considers exclusively integral values. Furthermore, it is mostly ignored that the values are actually bit-vectors in reality, so phenomena like integer overflow are not taken into account. If this is considered as a problem for an application using the algorithm, the analyzed control flow graphs must be modified before the analysis to include extra edges simulating these phenomena, for example by using modular arithmetic expressions.

1.3.3 Pure Expressions

This thesis will refer to ”pure expressions” at some points. A pure expression is considered to be an expression that does not change the program environment upon evaluation. A pure expression will also always evaluate to the same value as long as the program environment does not change.

1.3.4 SMT Solver

SMT is an acronym for satisfiability modulo theories. SMT is a decision problem over logical formulas. An SMT solver can be used to solve these formulas. In software verification, SMT solvers are often used to prove that a formula representing the reachability of an error location is unsatisfiable.

2 Background

This chapter elaborates on the theoretical background this thesis is based on. The first section describes the principles of k-induction, the second one shows the basics for generating invariants, and the third section explains the concept of configurable program analyses.

2.1 k-Induction

Mathematical induction is a well known method of proof usually used to prove assertions over the set of natural numbers. When trying to prove an assertion $P : \mathbb{N} \to \mathbb{B}$ over all natural numbers, the idea of mathematical induction is to prove $P(0)$ for the minimal element 0 of the set of natural numbers in the base step and then finding a proof that for every number $n+1$ following a number n for which $P(n)$ holds, $P(n+1)$ holds as well in the inductive step or, in other words,

$$(P(0) \wedge \forall n \, (P(n) \implies P(n+1))) \implies \forall n P(n)[25]. \tag{2.1}$$

k-induction is a generalization of k-induction using not only one, but k base cases in the base step and the assertion is required to hold for the k previous numbers in the inductive step. More formally:

$$A_k := \bigwedge_{i=0}^{k-1} P(i) \wedge \forall n \left(\left(\bigwedge_{i=0}^{k-1} P(n+i) \right) \implies P(n+k) \right) \tag{2.2}$$

where A_k is the k-induction principle and A_1 is the classic 1-induction explained above[25].

An important aspect of k-induction is that for any $k \in \mathbb{N}, k \geq 1, A_k \implies A_{k+1}$[25]. This means that the higher the value of k, the weaker the assertion used for the induction proof is, the reason being that in the step case there it considers not only the assertion over one predecessor $P(n)$ about the one number n before the next number $n+1$ but actually the assertion over k predecessors $\left(\bigwedge_{i=0}^{k-1} P(n+i) \right)$ which includes at least the assertion $P(n+k-1)$ but potentially more. This suggests that for k-induction with higher values of k, proofs might succeed provided with less additional information than for lower values of k.

The same induction principles can also be applied to the verification of program loops, where the assertion is not proven over the natural numbers but over the set of loop body executions, with the program state after the n-th loop body execution corresponding to the number n of the natural numbers in the mathematical induction methods shown above. For k-induction, the first k loop body executions can be unrolled. Proving an assertion to be true before each of those unrolled loop body executions as well as proving that for any k successive loop executions before each of which the assertion holds, the assertion holds before the $(k+1)$-st loop body execution, proves that the assertion is true before every execution of the loop body. Trivially, iff the assertion is true after any n-th loop body execution, it is true before the $(n+1)$-st loop body execution.

Consider a program containing a loop that supposedly is safe. When representing the program with a control flow graph, to prove that the program is safe it is necessary to show that no error location of the graph is reachable. Each path through the program graph can be represented as a formula by using the edge information as predicates. If all formulas of paths leading to error locations are unsatisfiable, the error locations are unreachable and the program is safe. Of course it is infeasible to enumerate all paths if a program contains a loop with a large or unknown number of iterations. Induction can be used to circumvent this problem. For programs with loops, it is necessary to prove the assertion that the formula of each path leaving the loop leading to an error location is unsatisfiable, or in other words, that there is no iteration through the loop so that a formula of a path exiting the loop in this iteration and leading to an error location is satisfiable. This assertion can be used as a safety property that when proven for each iteration of a loop, is sufficient to show that the loop is correct. When using k-induction to produce this proof, the safety property must first be proven for the first k iterations as the base case before proving that the safety property holds for any iteration n if it holds for its k preceding iterations $n-k$ to $n-1$ as the step case.

2.2 Basics of Invariant Generation

Assertions over the variables of a program that remain true whenever a certain program location is reached are called invariant assertions of a location[4]. The definition of inductive assertions is very similar, but more specifically tailored to loops in that an inductive assertion is an invariant assertion that holds the first time a location is reached, as well as every time the program loops back to the same location[4]. An assertion can be proven to be invariant by proving that a stronger inductive assertion can be found that implies the assertion to be proven [28, p. 91].

Invariant assertions for a loop thus provide information about the loop that can be helpful in proving other assertions, like showing that an error condition is never reached in any execution of the loop body, and may be used to support inductive proofs of loop correctness, such as strategies using k-induction.

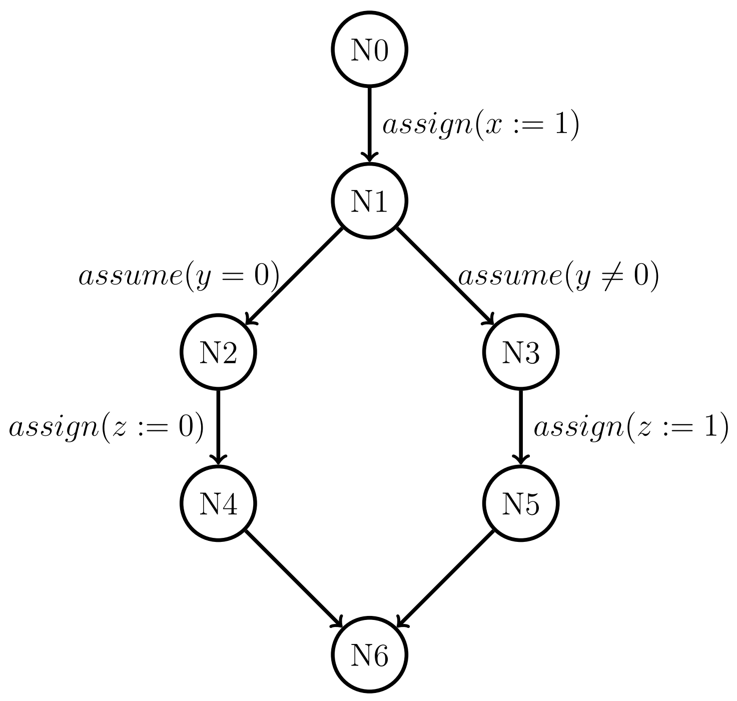


Figure 2.1: The control flow graph of a very simple program

Consider the control flow graph shown in figure 2.1.

It is easy to see that $x = 1$ at every node from $N1$ through $N6$. It is also obvious that $y = 0$ at $N2$ and $N4$ and that $y \neq 0$ at $N3$ and $N5$ as well as that $z = 0$ at $N4$ and $z = 1$ at $N5$. Each of these assertions is invariant at the respective mentioned locations.

If we add an assertion to the graph that leads to an error state if $x \neq 1$ in $N6$ as shown in 2.2, the invariant assertion $x = 1$ at $N6$ mentioned above trivially proves that the error state is unreachable.

Leaving it at that would mean that for $N6$ we only know that $x = 1$, which is a severe disadvantage when changing the error assertion of 2.2 to the more complex version in 2.3, because knowing that $x = 1$ at $N6$ does not suffice to prove that the error location is unreachable anymore. Looking a bit closer, we could, however, see that $z \in \{0, 1\}$ is also an invariant assertion at $N6$, because one of the two paths to $N6$ ensures $z = 0$ while the other path ensures $z = 1$, so depending on how $N6$ is reached, z must be either 0 or 1, and it is for example impossible for z to be 2 at $N6$. This information is sufficient to prove that the error location is unreachable in 2.3.

When modifying the error assertion again to be more complex as shown in 2.4, even the invariant assertions $x = 1$ and $z \in \{0, 1\}$ are insufficient to prove the error location to be unreachable. To disprove its reachability, we are required to retain the information that $z = 0$ iff $y = 0$ and $z = 1$ iff $y \neq 0$ as representable by an invariant assertion

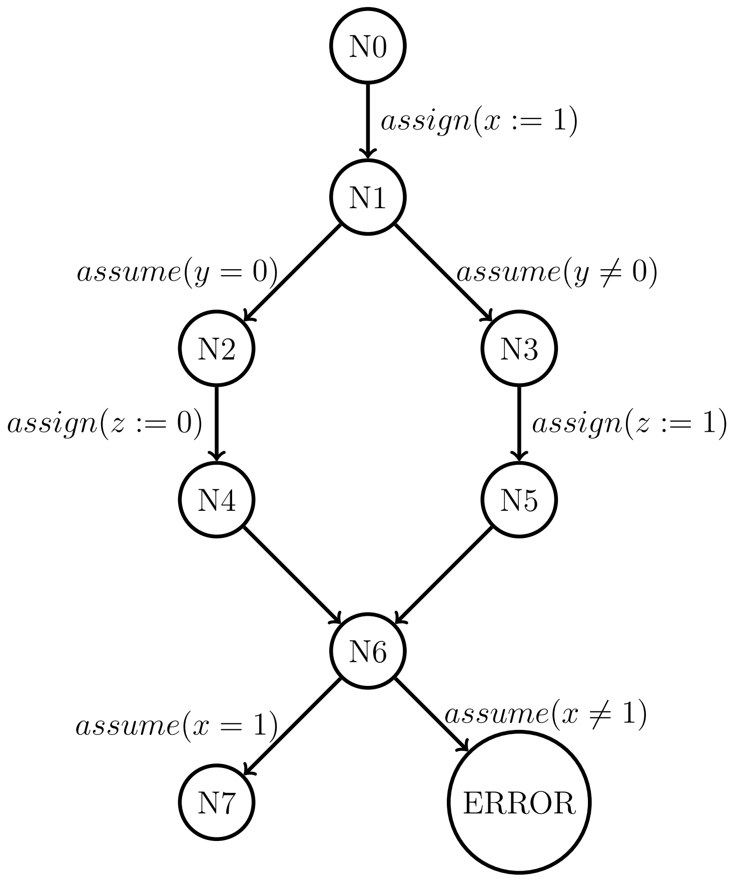


Figure 2.2: The control flow graph of a very simple program with an error assertion

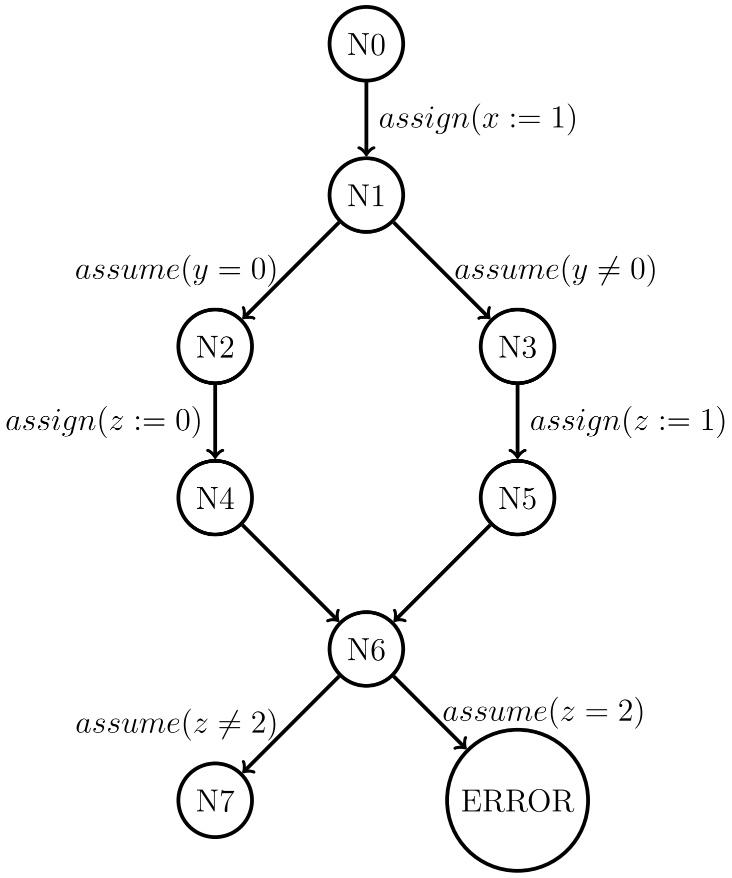


Figure 2.3: Another control flow graph of a very simple program with an error assertion

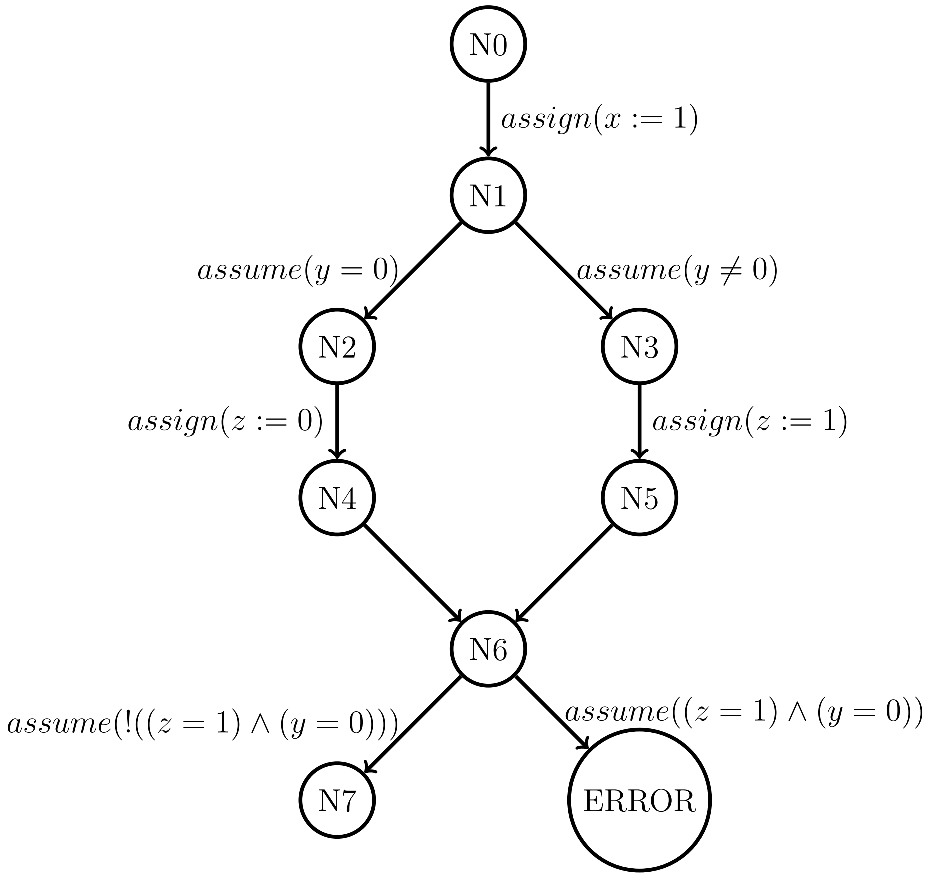


Figure 2.4: Another control flow graph of a very simple program with an error assertion

$((y = 0) \wedge (z = 0)) \vee ((y \neq 0) \wedge (z = 1))$ at $N6$. This information now covers the proof for the error location in 2.4 to be unreachable.

Such precision comes at a price, however. While it might be easy to collect this detailed information for small programs such as the one shown with various modifications in 2.1, 2.2, 2.3 and 2.4, it is infeasible for larger programs: n seemingly independent but consecutive `if-then-else`-decisions would already lead to 2^n different paths and as soon as a loop is contained the number of paths might even be infinite. This problem is, of course, a general one and in no way specific to generating invariants for k-induction, where k-induction in itself is an attempt to improve proofs over possibly infinite loops. Nevertheless it is a problem that must be dealt with, because while k-induction supposedly succeeds with weaker invariants than standard induction, proving non-trivial programs with k-induction mostly still requires non-trivial invariant assertions.

In fact, the question whether to trade performance for precision arises perpetually over the different aspects considered in order to create the invariant generation program implemented for this thesis. A wrong decision about any one aspect's performance could easily lead to infeasibility while reducing the precision too much might impede the ability to prove a correct loop. This section explains the decisions that coined the implementation presented in 4.

2.2.1 Information Extraction

There are basically two important types of control flow edges for information extraction: Assumption edges and assignment edges.

When a path through a control flow graph branches, the edges representing the different branches contain condition expressions that decide which way the actual execution of the program will chose. These edges are called assumption edges or assume edges. A pure assumption expression does not change the program environment, so after the transition over the edge, all program variables have the same value they had before the transition. The information that can be gained from an assume edge lies within the condition: Because the execution takes an assume edge if and only if its condition is met, we know that the condition holds at the successor of the assume edge.

An assignment edge represents the assignment of a value to a variable, where the value may be computed from an arbitrarily complex expression. Assuming that the expression itself is pure and ignoring language specific features like multiple aliases to the same memory location, the only variable that changes via an assignment edge is the assigned variable itself. The information that can be gained from an assignment edge is that the variable that was assigned to has the value of the assignment at the successor of the edge.

Just like multiple edges leaving one location splits one path into multiple paths, multiple edges leading to the same node occur. It is up to the analysis to decide whether to keep the different paths apart, thus preserving the differences in information as described in the last part of the introduction to this section, or treating the different paths as one single path from that point onward, thus losing the differences in information but potentially massively reducing the workload.

The raw extracted information also has the form of expressions, although they can often be simplified by combining them with each other. A formalized discussion of these methods of discovering and combining information about linear restraints among the variables of a program is given by Cousot and Halbwachs in their joint paper in 1978[18].

2.2.2 Information Representation

Over the course of analyzing a control flow graph to collect and refine invariants over the locations, sometimes information is added and sometimes information must be removed. Consider a control flow path where in the begin an assign edge $x := 1$ clarifies the variable x to have the value 1. If, later on the path, another assignment edge $x := y$ is encountered, any information we previously had about x is invalid and must be dropped, even the $x = 1$ collected earlier - unless, of course, y is known to be 1 as well.

It is often useful to be able to combine pieces of information collected from different locations, join information from different paths, extract the collected invariants to be used by a solver, or in some cases even determine a location to be unreachable without using an external solver. An important aspect for the efficiency of these operations is an efficient representation of the collected information. Sometimes efficiency may be bought at the cost of precision. In such cases the benefits and drawbacks of a solution must be discussed carefully, as each loss of precision might cause the verification to fail.

2.2.3 Abstract Interpretation

All of the previously shown example control flow graphs are free of loops. Given sufficient but finite time, it is possible for these programs to completely extract all information about all values of variables at all locations that is conveyed by the graphs and to represent this information in a finite amount of space, assuming that the chosen data structures are chosen wisely with respect to the aspects discussed in 2.2.2.

In contrast, consider the control flow graph shown in 2.5. As soon as a program contains loops, it generally becomes impossible to convert the graph into a representation like the one mentioned above without losing information. Not only might the number of iterations required to completely unroll a loop be extremely large, it might even be unknown, which leaves no other option but to consider it as potentially infinite if

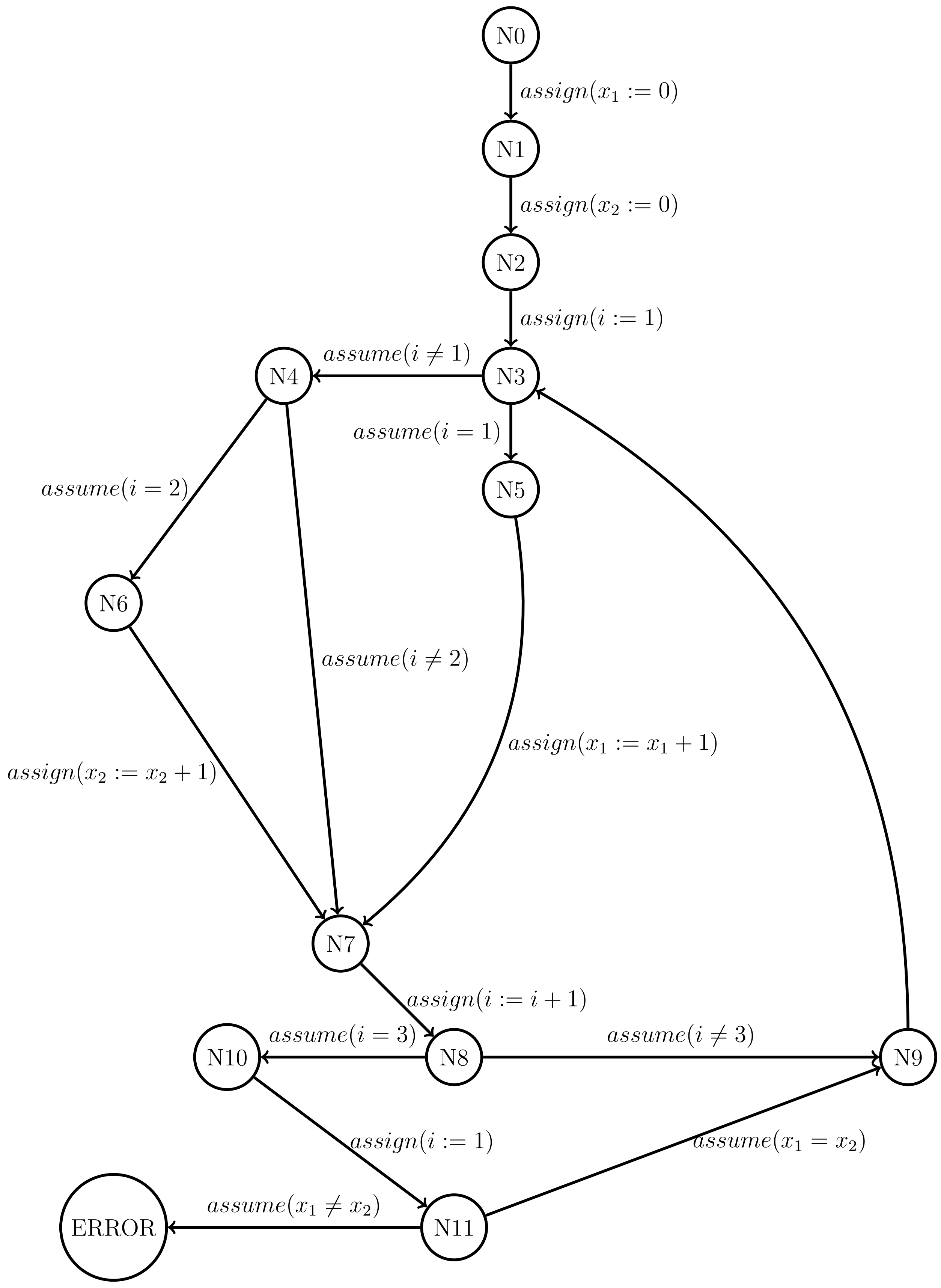


Figure 2.5: A control flow graph of a program containing a loop

no other information about the loop boundaries is known. Likewise, any attempt of completely unrolling such loops is futile.

Fortunately, the very idea of using k-induction is targeted at reducing the amount of information required for proofs, so, once again, we are able to reduce precision in favor of efficiency and termination. If the possible values of variables within a loop and the information about variable interrelations is approximated roughly enough, the analysis of the loop terminates, because after a finite amount of iterations through the loop, no new information is gained. The goal, of course, should be to keep said finite amount of iterations through the loops as small as possible while retaining as much information as possible. So instead of exactly interpreting the program code, we perform only an abstract interpretation.

Abstract interpretation, as presented by Cousot and Cousot in 1977[17], "is a general theory for approximating the semantics of discrete dynamic systems, for example, computations of programs" [29, p. 324].

In their initial example in their 1977 paper, Cousot and Cousot reduce variable values to their signs, leaving three possible states for a variable: Positive, negative, or unknown, denoted as $\{+, -, \pm\}$ [17, p. 238]. As this, of course, is just an example for the general idea of using abstract representations covering large parts of state spaces, using an unmodified version of this strategy might cause issues. For example, it raises the question how to deal with the value 0. It could be included in $+$ or $-$ or simply be treated as $\pm$. While it is certainly possible to use an abstract interpretation treating 0 as either of these values by taking this special case into account, it seems not only confusing to represent a value that is neither negative nor positive by $+$, $-$, or $\pm$. Supposing that 0 is a significant value in many programs, as might be suggested by programming languages like C treating it as the boolean `false` as well as 0 being the size of any kind of empty collection, it might even be disastrous to the precision of an analysis if the value 0 is indistinguishable from all positive or all negative values or is even automatically treated as an unknown value. Thus, it might be desirable for an analysis to handle 0 values in an extra case.

But even that might often still not be enough precision. Fortunately, a more precise version of abstract representation will be discussed in 3.2.1.1: Intervals or lists of intervals. They do allow for representing signs by intervals from negative infinity to -1 or 1 to positive infinity, the unbounded interval covers the unknown case and even 0 is easily represented by a singleton interval from 0 to 0. Moreover, intervals may be used to cover a lot more cases if required for precision.

Representation, however, is only one aspect of abstract interpretation. When analyzing an expression over concrete or abstract values, an abstract result needs to be found. The easiest but least precise way to compute the result of an expression would be to define the value of every expression as unknown. While this approach is of course useless for an analysis in general, it can be used as a fallback option in cases where the computation of a more precise result is too complicated. It is important to keep in mind that the very reason why abstract interpretation is used is to avoid too high precision. Thus, it must be carefully decided exactly how precise the evaluation of an expression should be at any point during the analysis.

There is, on the one hand, no harm in exactly evaluating expressions that are not part of any loop. Diverging paths can be merged back together when they reconnect or treated separately as already discussed in 2.2.1. Expressions that are part of a loop, on the other hand, require more sophistication. Ideally, the abstraction would cover no more than the variable values actually possible within the loop, but since often not even the number of iterations through a loop is known and the chosen representation only allows for a finite amount of gaps between value ranges, it is in general futile to pursue this goal.

2.3 Configurable Program Analysis

Configurable software verification is a concept researched by Beyer et al. Specifically, Beyer et al. designed a strategy called configurable program analysis or `CPA`[24]. One of the goals of this strategy is to facilitate the combination, configuration and comparison of different program analyses. Another benefit of their software verification framework CPAchecker built around this strategy is that new analyses can be implemented more quickly and in a standardized way[9], as many tasks often required for program analysis, like SMT solvers or parsing code and transforming it into control flow automata, are already available within the framework.

The following paragraphs paraphrase the definition of a configurable program analysis given by Beyer et al. in their 2007 paper about configurable software verification[24].

A configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, \mathit{merge}, \mathit{stop})$ is defined by an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a termination check stop [24, p. 506].

It is possible to further break down the definition of the abstract domain into $D = (C, \mathcal{E}, [\![\cdot]\!])$, where C is the set of concrete states the abstract domain consists of, $\mathcal{E}$ is a semi-lattice and $[\![\cdot]\!]$ is a concretization function. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of the set E of abstract elements, elements for top $\top$ and bottom $\bot$, where $\top \in E$ and $\bot \in E$ as well as a preorder $\sqsubseteq \subseteq E \times E$ and a join operator $\sqcup : E \times E \to E$ which is a total function. $[\![\cdot]\!] : E \to 2^C$ is the concretization function assigning a meaning to an abstract state by expressing which concrete states it covers [24, p. 506].

The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ is used to assign abstract successors e' to a state e. g denotes the control flow edge corresponding to the transfer [24, p. 506 f.].

The merge operator $\mathit{merge} : E \times E \to E$ is used to combine information of abstract states. Because both functions have the same input and output sets, the merge operator seems similar to the join operator $\sqcup$ defined for the semi-lattice of the abstract domain. However, the two operators are applied to achieve different goals. The merge operator may be based on the join operator, but not necessarily is. For soundness of the analysis, it is required that the result of the merge operator is at least as as abstract as its second parameter [24, p. 507].

Lastly, the termination check $stop : E \times 2^E \rightarrow \mathbb{B}$ is an operator used to check whether or not the set of abstract states given as the second parameter covers the abstract state given as the first parameter. Just the way the merge operator seems similar to the join operator, the termination check seems similar to the preorder $\sqsubseteq$ of the lattice and again, the similarity is misleading, as termination check and preorder differ in the way they are used. Nevertheless, the termination check can be based on the preorder [24, p. 507].

The actual configurable program analysis algorithm is given in pseudocode by Beyer et al. [24, p. 508]:

```
Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$, an initial
       abstract state $e_0 \in E$ where $E$ denotes the set of elements of the
       semi-lattice
Output: a set of reachable abstract states
Variables: a set reached of elements of $E$, a set waitlist of elements of $E$
waitlist := $\{e_0\}$
reached := $\{e_0\}$
while waitlist $\neq \emptyset$ do
    pop $e$ from waitlist
    for each $e'$ with $e \rightsquigarrow e'$ do do
        for each $e'' \in$ reached do do
            // Combine with existing abstract state
            $e_{new} := merge(e', e'')$
            if $e_{new} \neq e''$ then
                waitlist := (waitlist $\cup \{e_{new}\}) \setminus \{e''\}$
                reached := (reached $\cup \{e_{new}\}) \setminus \{e''\}$
            end
        end
        if $\neg stop(e'$, reached$)$ then
            waitlist := waitlist $\cup \{e'\}$
            reached := reached $\cup \{e'\}$
        end
    end
end
return reached
```

Algorithm 1: The CPA algorithm taken from Beyer at al. [24, p. 508]

The algorithm takes a configurable program analysis as defined above and an initial abstract state $e_0 \in E$ of the semi-lattice D as its input. During the algorithm, two sets containing abstract elements are used, one of them representing states that have been reached by the analysis E_r and another set that represents the states waiting to be processed E_w. Initially, the initial state is added to the set of waiting states. In every iteration of the outer loop, one state e is taken from the set of waiting states and is processed. This processing involves using the transfer relation $\rightsquigarrow$ to obtain the abstract successors of e. Each of those abstract successors e' is then merged with each state $e'' \in E_r$ of the set of reached states to obtain the merged state $e_{new} = merge(e', e'')$. If $e_{new} = e''$, the merge result did not result in an abstraction, as the second parameter is the least abstract state the merge operator is allowed to produce. If $e_{new} \neq e''$, the new state e_{new} is added to both the set of reached states E_r and the set of states waiting to be processed E_w. Independently of the result of the merge operation, the stop operator is invoked on the abstract successor e'' and the set E_r of reached states. If it the result of the stop operation is negative, e'' is added to E_r and E_w. In any case, the next iteration of the outer loop of the algorithm then begins, supposing $E_w \neq \emptyset$. Otherwise, the algorithm terminates and gives E_r as its output [24, p. 508].

3 Theory

In this chapter, the theory this thesis is based on is explained. The first section explains how the generated invariants are combined with k-induction. The second section focuses on the techniques used to accomplish the task of generating invariants chosen for this thesis. The third section shows how these techniques are used to define a configurable program analysis for generating invariants.

3.1 Providing k-Induction with Invariants

2.1 explained the properties and benefits of k-induction, but also stated that for induction proofs to succeed, information about the analyzed programs is required. The control flow automaton shown in figure 2.5 represents an actual code example tested with the implemented algorithm. If the bounded model checking algorithm implemented in CPAchecker applies its induction proof to this example, it fails even for $k = 100$. Provided with the invariant $(i = 1 \land x_1 = 0 \land x_2 = 0) \lor ((i \in \{1, 2\} \lor i \geq 4) \land x_1 \geq 1 \land x_2 \geq 1)$ by the implemented invariant generation algorithm, however, the verification succeeds for $k = 2$. To understand how information is supplied to the induction, knowledge about the how the induction proof is constructed is required.

As the base case of the induction can easily be handled by enumerating the contained iterations and checking the paths leaving the loop in these iterations is usually feasible, the base case is trivial. When using bounded model checking with a bound k, the task of proving the base case is already covered. The step case is more complicated. It can be formulated as consisting of three parts A, B and C when written as $(A \land B) \implies C$, where C is the assertion that the safety property holds for any iteration n, B is the condition stating that the safety property holds for the k predecessors, and A is an additional condition provided by invariant assertions. Thus, if $(A \land B) \implies C$ is a tautology, the program is safe. This can easily be checked by using an SMT solver to prove that $\neg((A \land B) \implies C)$ is unsatisfiable. The invariant generation algorithm implemented for this thesis is responsible for providing part A of this formula.

But while experiments show that k-induction works well for some programs, other examples benefit less from the technique [8, p. 367]. To the knowledge of the author, as yet, no clear characterization of the classes of programs that benefit most from k-induction has been determined. The fact that k-induction is superior to classic induction by considering - and thus differentiating between - different base cases, however, suggests that k-induction might be most useful for verifying programs containing loops with multiple loop body cases.

3.2 Invariant Generation

2.2 explained the theoretical background of invariant generation used for this thesis. This section shows the approaches selected for the implementation.

3.2.1 Efficient Representation

2.2.2 discussed the requirements on information representation and addresses that sacrificing precision to gain verification speed is sometimes beneficial, but has the drawback of jeopardizing the verification success.

3.2.1.1 Sets of Integer Values

As shown in 2.2, there are cases like the one presented in figure 2.3 where it is sufficient to narrow down the possible values for a specific variable. It is thus desirable to be able to represent the possible values of a variable as a set. When dealing with large or possibly infinite amounts of elements, however, one has to consider that set implementations which use explicit representations of every element contained are not applicable.

A simple way of representing large or even infinite sets of integers in a constant amount of space is the interval, the drawback being that they cannot contain gaps. An interval i consists of a lower bound lb and an upper bound ub, where $lb \leq ub$. The boundaries of the interval lb and ub are considered to be included in the interval, unless they are infinite. Inclusive bounds are denoted with brackets, exclusive bounds with parentheses, so the interval from zero to positive infinity becomes $[0, Inf)$. For ease of notation, in cases where the value of a bound is not specified in this thesis, it is denoted as if it were known to be inclusive, unless the difference is significant to the topic.

It is possible to represent the range of values from -2^{24} to 2^{24} by just stating those two bounds in an interval $i := [-2^{24}, 2^{24}]$, but due to the inability of intervals of representing gaps in the set it is impossible to represent said range without the value 0 with just one interval. Another problem that might arise with intervals is when trying to represent an empty set. This second issue, of course, might be easily fixed with a special convention or an additional flag without impairing the general idea of using intervals.

Another method to represent sets of integers is using lists of intervals. Such lists of intervals now allow for gaps in the sets and with the empty list provide an easy way to represent the empty set. Lists of intervals, however, again suffer the initial problem of possibly growing linearly in size with the amount of values represented, for example when considering the infinite set of even numbers. Nevertheless, they still allow for representing all sets representable by a single interval in a constant amount of space while at the same time enabling the representation of more complex sets. It is then up to the application employing this set representation to use it in a responsible way.

Considering operations on the sets, there are differences between single intervals and lists of intervals as well. Checking whether or not a value is contained is a constant operation for single intervals, but logarithmic in the amount of intervals in a sorted list of non-overlapping and non-touching intervals. Two intervals $i_1 := [lb_1, ub_1]$, $i_2 := [lb_2, ub_2]$ do not overlap if $lb_1 > ub_2 \vee ub_1 < lb_2$, or in other words, if the intersection of the sets represented by the two intervals is the empty set. Two intervals $i_1 := [lb_1, ub_1]$, $i_2 := [lb_2, ub_2]$ are considered as non-touching if the intervals do not overlap and $min(|lb_2 - ub_1|, |lb_1 - ub_2|) > 1$, or in other words, if there is at least one value between the intervals that is included in neither of them. Ensuring that intervals to not overlap or touch in turn complicates the union of sets, which is linear in the combined amount of intervals of both sets, but is constant if using only single interval sets at the cost of a loss of precision when spanning from the lowest lower bound to the highest upper bound of the sets being united.

So while contrary to using lists of intervals, using single intervals to represent the sets of values is simpler, more time efficient and more space efficient, the list of invervals provides a much wider range of possibilities and thus, higher precision. But not only do lists of intervals allow for a higher precision, if desired they can be used just like the single intervals by using lists with just one element each. For the implementation of the invariant generation algorithm, the choice therefore fell on representing the sets with lists of intervals.

These lists of intervals and their potential for representing large sets can be visualized with some simple figures. Figure 3.1 shows a list of intervals with one element representing all integer values. Figure 3.2 presents an empty list of intervals representing the empty set of integers. Figure 3.3 is a visualization of the solution to the problem of not being able to represent gaps in sets with single intervals, such as representing all integer values without the value 0. Figure 3.4 shows a list of intervals with two finite elements, thus representing a finite set of integer values with a gap between the intervals. Lastly, figure 3.5 shows another list of intervals containing three elements, but while two of the elements are finite, the third one is infinite. Note that some of the intervals are already visualized with a lattice in mind: While the lists of intervals themselves are simply sets of integer values, they are used to describe the possible values of expressions or variables. So of course figure 3.5 shows a set containing the values -10, -9, -8, 0, 10, and every value greater than 10, but when used as the representation of possible values of a variable, there is only one actual integer value

All integer values, may be used to represent $\top$ in a lattice
(-Inf, Inf)

Figure 3.1: A list of intervals representing $\top$

$\emptyset$, may be used to represent $\perp$ in a lattice

Figure 3.2: An empty list of intervals representing $\perp$

of the variable that is not known exactly but might be either -10, -9, -8, 0, 10, or any value greater than 10.

3.2.1.2 Variable Environment

When analyzing a program, there is usually more than one variable and not only the specific values a variable might be of interest but also any relations between different variables. Sometimes for example two variables might be known to be equal even though there is no information about their actual value. It is therefore not always sufficient to keep a mapping between variables and their corresponding sets of possible values. Unless the specific value of two equal variables is known, much more information would be preserved if for example instead of storing a set of possible values for both variables, the set is stored for only one of the two variables while the second one links to the first one.

This idea, however, does not take inequalities into account. Two variables might be known not to be equal, one might be known to be larger than the other one, possibly even by a known value. The logical consequence would be not to map sets of possible values to variables, but rather formulas that might contain such sets but might also represent complex relationships between variables.

Anything but 0	
(-Inf, -1]	[1, Inf)

Figure 3.3: A list of intervals representing all integer numbers but zero

2 to 4, 8 to 10	
[2, 4]	[8, 10]

Figure 3.4: A list of intervals with two finite intervals

The values -10, -9, -8, 0, 10, any value greater than 10		
[-10, -8]	[0,0]	[10, Inf)

Figure 3.5: A complex list of intervals

But switching the way of representing information about a variable from sets of possible values to complex formulas again rises the questions discussed in the introduction to this subsection. The more complex such a formula becomes, the more complex it becomes to modify the information about the variable, because the whole formula must be analyzed and changed accordingly. The invariant generation algorithm using such a mapping of variable names to expressions is therefore required to control the complexity of the expressions so as not to risk negative impacts on performance.

The mapping makes it easy to quickly access an expression associated with a variable and can thus be used to evaluate a variable by repeatedly replacing other variables contained in its expression with their respective expressions until no variables remain. This, however, is only possible if the mapping does not contain loops. This might seem trivial at first, but sometimes it is difficult to decide to what variable an expression that could as well be resolved to a different variable is supposed to be stored. To avoid loops in the mapping, such expressions must sometimes be simplified, and this simplification can potentially reduce the precision.

To mitigate this problem, the environment is accompanied by a very simple set if variable interrelations. These interrelations are binary relations over variables using the operators of the set O of operators where $O = \{=, \neq, <, \leq, >, \geq\}$ so that for each set S of these sets, $S \subseteq V \times O \times V$, where V is the set of variables.

3.2.1.3 Path Distinction

The control flow graph in figure 2.4 shows an example for a program where simply knowing the possible values of a program variable is not sufficient to prove the program correct. Rather, it is sometimes necessary to know under what conditions which values are possible. A mapping from variable to value or expression as suggested in 3.2.1.2 is not the proper data structure for such information.

In the previously mentioned example, the solution shown was to keep the paths apart, but the description also explained the general infeasibility of that approach. It is thus not sensible to try to analyze every program path separately, while on the other hand the distinction of at least partial paths is required for some proofs.

Using a configurable program analysis for the invariant generation algorithm allows for generating multiple states for the same program location, which can be used to separate paths by generating different invariant assertions for different paths and treating those states as a disjunction of the paths. There is, however, also the option of merging states by using the merge operator defined in 2.3, so that the information of the merged states can be combined [24, p. 505]. In the case of the invariant generation algorithm, this means dropping the distinction between the merged paths.

This leads to the problem of deciding when to merge states and when to keep them apart, or in other words the definition of the join operator, supposing the merge operator $merge^{join}$ is used. Merging too often results in a loss of too much information to prove correctness, while merging too few states leads to an explosion of states as described by Clarke et al. [6, p. 79]. Ideally, the algorithm should always merge two states if dropping the distinguishing information is irrelevant to the success of verification and never merge them otherwise. Unfortunately, there is no reasonable way of deciding in advance exactly which information is really required.

Some information, of course, may be eliminated as irrelevant immediately, for example any variables that do not appear in any assume edges of paths leading to the error location. Conversely, variables that do appear on such assume edges might be considered as relevant, but not necessarily are: consider the example shown in 2.3, where the value of y is in no way relevant for the proof, while in 2.4 it certainly is. Multiple strategies for deciding when to merge and when not to are conceivable.

One idea is to analyze which variables appear more often in assume edges leading to an error location and which variables appear less, prioritizing information retention for the more frequent variables and merging states when the dropped information affects mostly less frequent variables.

Another option is ranking the variables appearing in assume edges by their proximity to the error location instead of by their frequency. This strategy is specifically useful in cases where the assume edges around the error location are designed to specify the actual error condition represented by the error location.

Of course, many more different strategies are conceivable. It has to be evaluated over extensive benchmarks which strategy provides the best results in practice. Figure 3.6 shows an example program where both of the strategies mentioned previously produce a sub-optimal relevance ranking for the variables: Both strategies would rank x most important, because it appears on the most assume edges leading to the error location as well as on the assume edge closest to the error location. x, however, is 1 on every path leading to the error location anyway. It would be more beneficial to closely analyze the variables y and z, as their possible different constellations are relevant for the proof.

The option chosen for this thesis is the second of the strategies described above: Ranking the variables appearing in assume edges leading to the error location by their proximity to the error location and prioritizing information about the closer variables as opposed to information about variables farther away.

3.2.2 Expression Evaluation and Abstraction

At many points during the analysis, it is required to evaluate collected formulas to an over-approximation of the values covered by formula: Formulas can be partially simplified by evaluating constant parts or even completely replaced by the over-approximation to increase the amount of represented state space by decreasing the precision, and a formula that evaluates to an empty set of values is unsatisfiable, which allows for some optimizations. While it might therefore initially seem desirable to always evaluate expressions as exact as possible, 2.2.3 discussed the necessity of abstract interpretation and an approach to its implementation.

The option chosen for this thesis is evaluating expressions in loops as exactly as possible, but storing the information in a way similar to the example strategy of Cousot and Cousot, focusing on signs, with slight modifications for some special values. Thus, the information gain from expressions in loops is rather imprecise, but as has been elaborated in 2.2, this lack of precision is necessary to ensure not only efficiency but even termination of the analysis.

It has already been discussed in 2.2.3, that while a high precision is desirable for the quality of the results of the analysis, lower precision is sometimes required to guarantee the termination of the algorithm. Therefore two strategies for evaluation formulas are implemented: One strategy tries to achieve a high precision by evaluating expressions as exact as possible, while the other strategy deliberately generates much less precise results. For example, the first strategy would evaluate $x = 1 + 1$ to $x = 2$, while the second strategy might produce $x > 1$. For the aforementioned reasons, it is vitally important to the algorithm that this abstract evaluation strategy produces such imprecise results. The abstract evaluation strategy first evaluates each expression like the exact evaluation strategy, but then abstracts the result r to the abstract result $r_a := \bigcup\{p|p \in \{\{(-Inf, -1]\}, \{[0, 0]\}, \{[1, Inf)\}\} \wedge p \cap r \neq \emptyset\}$. This way, the value 0 as well as the set excluding it, {(-Inf,-1], [1, Inf)}, can still be represented exactly, while at the same time allowing sign based abstraction as discussed in 2.2.3 with only the exception of the value 0. The set of exceptions, namely $\{0\}$, could easily be extended to further increase precision; such an extension however would once again lead to a lowered performance as more different abstractions are possible.

This concept of abstraction with exceptions for set representations is very similar to the concept used to over-approximate states on merging based on potentially interesting

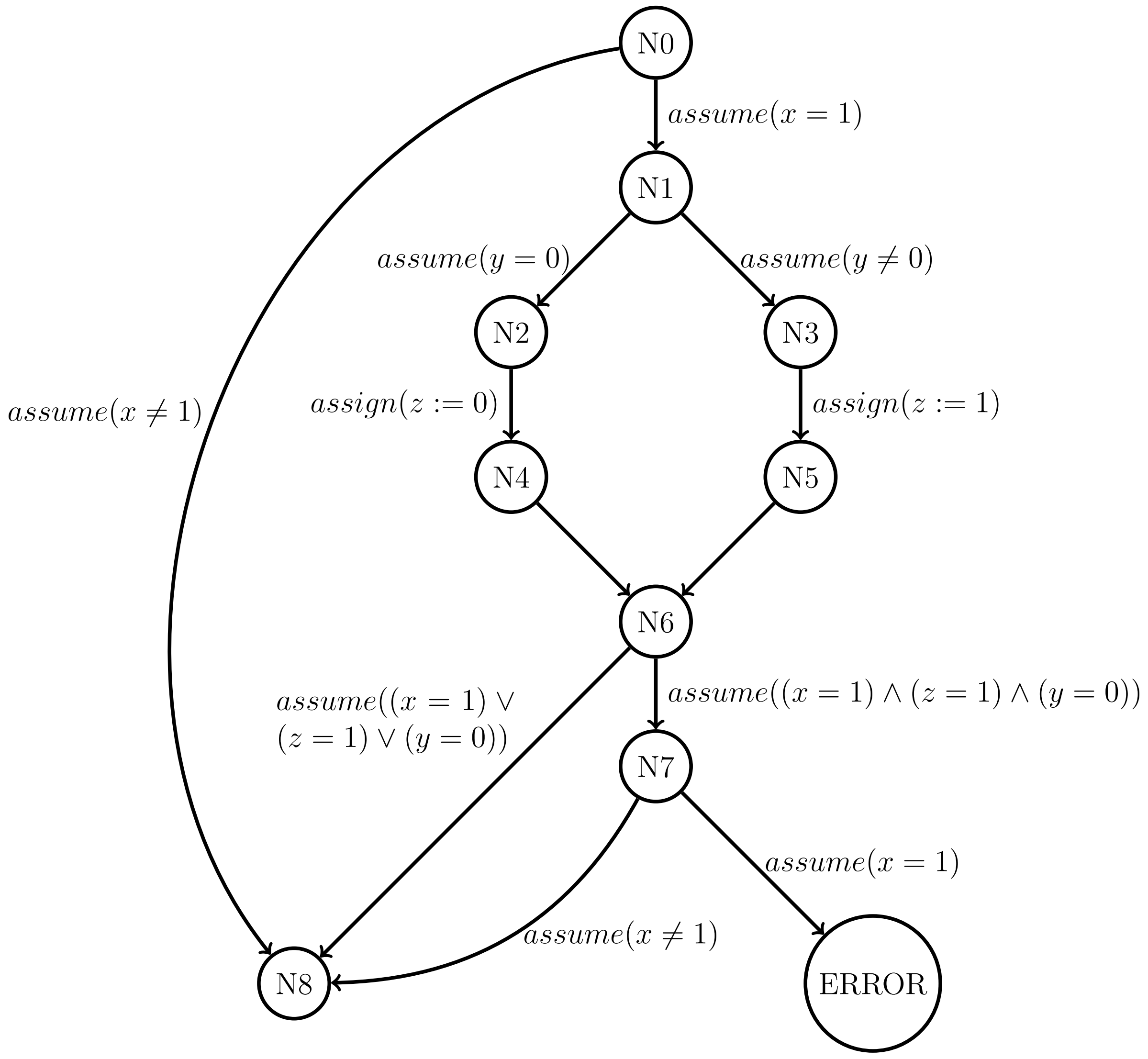


Figure 3.6: An example control flow graph of a program where variable ranking might be sub-optimal

variables as in the heuristic described in 3.2.1.3. A possible extension to this strategy for over-approximating the sets could be made by extending the set of exceptions to the abstraction by guessing which constants might be interesting to the analysis, for example by performing a pre-analysis to determine constants on assume edges close to the error locations. This idea, however, has not yet been implemented and therefore no evaluation results can be presented for it.

3.3 A Configurable Program Analysis for Generating Invariants

As the generated invariants will be used by the bounded model checking algorithm implemented in CPAchecker, the probably most straight-forward approach to integrate invariant generation is to define it as a configurable program analysis or `CPA`. The properties of a `CPA` were defined in 2.3.

The invariant generation `CPA` is hereby defined as $\mathbb{I} = (D_i, \rightsquigarrow_i, merge_i, stop^{sep}$ where $merge_i \in \{merge^{join}, merge^{sep}\}$.

These merge operators $merge^{join}$ and $merge^{sep}$ as well as the stop operator $stop^{sep}$ are taken from Beyer et al. and can be found in their 2007 paper on configurable software verification[24].

$merge^{join}$ is a merge operator based on the join operator, so that $merge^{join}(e, e') = e \sqcup e'$ and $merge^{sep}$ is a merge operator that always returns the second parameter so that $merge^{sep}(e, e') = e'$, thus no merged states are ever used by the configurable program analysis algorithm [24, p. 507]. While $merge^{sep}$ preserves the most information as no precision is lost by merging different states, it also cannot benefit from the advantages of merging states in time and memory consumption described in 3.2.1.3. The solutions presented there only apply when $merge^{join}$ is used, which therefore is the default option as will become apparent in table 4.1.

$stop^{sep}$ is a stop operator based on the preorder of the semi-lattice, so that $stop^{sep}(e, E_r) = (\exists e' \in E_r : e \sqsubseteq e')$ [24, p. 507].

The abstract domain $D_i = (C, \mathcal{E}_i, [\![\cdot]\!]_i)$ consists of the set C of concrete states, the set $\mathcal{E}_i$ of abstract states and the concretization function $[\![\cdot]\!]_i$ where $\mathcal{E}_i = (E_i, \top, \bot, \sqsubseteq_i, \sqcup_i)$ is a lattice based on states where every abstract element or state $e \in E_i$ represents the collected information about invariants. This information is represented as a mapping of expressions and an accompanying set of relations between variables as described in 3.2.1.2. These expressions may consist of other expressions, variables or constants. As the actual values of the constants are not always known, they are modelled by the sets described in 3.2.1.1, where a set represents the information that the modelled constant is a value contained in the set. An abstract element with no information is the state$\top$, an abstract element containing a contradiction in the represented information represents the state $\bot$, for example if any expression mapped to a variable evaluates to an empty set of possible values.

$E_i \subseteq M \times R$ where M is the mapping of expressions to variables and R is the set of binary variable interrelations. The mapping M of expressions to variable is defined as $M : V \rightarrow \mathit{Expr}$, where Expr is the set of expressions and V is the set of variables. The set Expr of expressions is defined recursively as $\mathit{Expr} \subseteq ((\mathit{Expr} \times B \times \mathit{Expr}) \cup (U \times \mathit{Expr}) \cup V \cup 2^{\mathbb{Z}})$. In this definition, V is again the set of variables, $\mathbb{Z}$ is the set of integer values, so that a constant can be modelled by a set of integer values, U is the set of supported unary operators $U = \{\neg, \sim, -\}$ and B is the set of supported binary operators $B = \{+, *, /, \%, =, <, \hat{}, |, \vee, \&, \wedge, >>, <<, \cup\}$. As described in 3.2.1.2, given the set O of comparison operators where $O = \{=, \neq, <, \leq, >, \geq\}$ and the set V of variables, the binary variable interrelation set is defined as $R \subseteq V \times O \times V$.

Subsequently, the preorder function $\sqsubseteq_i \subseteq E_i \times E_i$ is defined so that $\sqsubseteq_i (e, e')$ evaluates to `true` iff the information represented by the abstract element e implies the information represented by the abstract element e' and there is no difference in information regarded as especially interesting according to 3.2.1.3.

The join operator $\sqcup_i : E_i \times E_i$ is defined so that $\sqcup_i(e, e')$ produces an abstract element e'' representing an over-approximation of the disjunction of the information of abstract element e and abstract element e', but only if the abstract elements are not considered to be incompatible for merging by differences in information regarded as especially interesting according to 3.2.1.3. In the latter case, e' is returned and the abstract elements are not joined. As mentioned above, this only affects the behavior of the configurable program analysis algorithm if $merge^{join}$ is used.

The definition of the transfer relation $\rightsquigarrow_i \subseteq E_i \times G \times E_i$ is that for a predecessor $e \in E_i$ a control flow edge $g \in G$ leaving e, the successor is e if the edge g does resemble neither an assignment nor an assumption edge. Otherwise, the information of the successor state e' is a modification of the predecessor state e where the information is changed according to the assumption or assignment made on the edge g as described in 2.2.1 by using the techniques described in 3.2. So $\forall g \in G : (\exists e, e' \in E_i \wedge g \notin (G_{assume} \cup G_{assign}) \implies e = e'$, where $G_{assign} \subseteq G$ is the set of assignment edges and $G_{assume} \subseteq G$ is the set of assume edges.

4 Implementation

The invariant generation is implemented as a configurable program analysis within the software verification framework CPAchecker [9] and is used by the bounded model checking implementation of CPAchecker.

Like every configurable program analysis in CPAchecker, the InvariantsCPA consists of abstract elements, an abstract domain and a transfer relation. It uses the $stop^{sep}$ operator and by default also the $merge^{join}$ operator, as described in 3.3.

As stated in 1.3.2, the presented algorithm is limited to integral values, not floating point values and also not bit-vectors. It has also already been explained that therefore phenomena like integer overflow are not handled by the analysis. Although the present implementation encapsulates the affected value representations in a way that is designed to make them exchangeable in case of future improvements, it is currently limited by this restriction.

4.1 Overview

It has already been mentioned above that the invariant generation algorithm is implemented as a configurable program analysis within CPAchecker. 3.3 explains the properties of the components required in order to use a `CPA` to implement the invariant generation algorithm. The main class implementing the definition of the `CPA` is the class InvariantsCPA described in 4.2.

One important component required for the definition of the `CPA` is the definition of the abstract elements described in 3.3. The abstract elements are implemented in the InvariantsState class described in 4.7. The environment mapping and the accompanying set of additional variable interrelations storing the information represented by the InvariantsStates consist of expressions which in turn are implemented by the InvariantsFormula class described in 4.6. An InvariantsFormula can be any type of the implemented unary or binary operations over other InvariantsFormulas or a variable or a constant. It has already been explained that the actual values of the constants are not always known, and that they therefore are modelled by the sets described in 3.2.1.1, where a set represents the information that the modelled constant is a value contained in the set. These sets are implemented by the class CompoundState depicted in 4.5. As explained in 3.2.1.1, the set representation chosen for this thesis uses sorted lists of intervals. These intervals are implemented by the class SimpleInterval presented in 4.4.

Another important component is the transfer relation implemented in the InvariantsTransfer relation described in 4.8. The transfer relation is responsible for computing the abstract successors corresponding to a control flow edge leaving a state represented by an abstract element. As mentioned above, these abstract elements are represented by instances of the class InvariantsState.

A further component required for the definition of the `CPA` is the abstract domain implemented by the class InvariantsDomain depicted in 4.3. It contains the join operator and the preorder.

4.2 InvariantsCPA

The class InvariantsCPA defines the configurable program analysis used to implement the invariant generation algorithm. It manages the merge and stop operators, the abstract domain InvariantsDomain, the configuration and the initialization.

4.2.1 Initialization

To start the configurable program analysis, an initial state is required. The function for computing this initial state is part of the InvariantsCPA class. The following initialization procedures aggregate data used to construct the initial state and are implemented in this function.

Before the actual algorithm runs, it is automatically initialized with some data affecting the analysis. First, all edges on paths leading to an error location are collected. All other edges are irrelevant for proving an error location to be unreachable. This is an optimization that could also be implemented by removing all irrelevant paths from the control flow graph prior to the analysis, so this strategy might be removed in the future to provide a cleaner, more generic algorithm. Until then, the optimization can be switched off via a configuration flag, but defaults to being turned on. If the optimization is turned off, all edges are considered to be relevant.

The optimization can be improved even further by analyzing what variables appear on the relevant assume edges and what other variables they are influenced by. As only those variables are relevant for proving the unreachability of an error location, others need not be analyzed. Again, this optimization could be replaced by stripping appearances of irrelevant variables from the control flow graph before starting the invariant generation algorithm. For the time being, this improved optimization if performed by default, but may be switched off via a configuration flag. If the optimization is turned off, all variables are considered to be relevant.

Another element of the initialization process is the guessing of interesting information to be used for deciding about the merging of states as described in 3.2.1.3 and 4.7.3. The guessing may include variables, and it may include generic predicates, depending on the configuration. By default, the guessing of variables is enabled, while the guessing of generic predicates is disabled. To guess what information might be interesting, the algorithm starts a backwards breadth first search through the control flow graph starting from the target locations. When an assume edge is found, its expression is a candidate to be interesting information. Based on the configuration, it might be used as a generic interesting predicate, its variables might be used as interesting variables, or it could be ignored if the limits of interesting predicates and variables are already reached. Predicates that are detected as already covered by the guessed variables are ignored and do not count toward the limit.

Variables considered to be interesting are a subset of the set of variables that are considered to be relevant that was described above. These sets should not be confused with each other.

4.2.2 Configuration

The algorithm can be configured via the configuration options shown in table 4.1. As there are default values for all options, none of them has to be set explicitly. There are further important configuration options that do not directly affect the implemented algorithm, but are used to enable its usage in other analyses or make the algorithm more useful for the analyses using it. These parameters are listed in table 4.2. All the default values shown are default on an application wide basis for CPAchecker; naturally these default values are sometimes overridden by standard configurations of certain strategies. A notable example for this is the standard configuration for bounded model checking with induction, where `bmc.induction` is set to `true`, `bmc.useInvariantsForInduction` is set to `true`, `cpa.predicate.solver.useIntegers` is set to `true` and `cpa.loopstack.maxLoopIterations` is set to `1`. `cpa.loopstack.maxLoopIterations` of course must always be overridden with the desired value of k for k-induction.

4.3 InvariantsDomain

The InvariantsDomain class maps to the abstract domain described in 3.3. It implements the join operator and the preorder of the lattice. As both aspects require intricate knowledge of the InvariantsState class that is designed to be encapsulated, the actual implementation of these aspects is delegated to the InvariantsState class.

Name	Description	Default
`cpa.invariants .analyzeTargetPathsOnly`	Determine target locations in advance and analyze paths to the target locations only. See 4.2.1.	`true`
`cpa.invariants .analyzeRelevantVariablesOnly`	Determine variables relevant to the decision whether or not a target path assume edge is taken and limit the analysis to those variables. See 4.2.1.	`true`
`cpa.invariants .interestingPredicatesLimit`	The maximum number of predicates to consider as interesting. `-1` one disables the limit, meaning that all predicates are considered to be interesting, but this is not recommended. `0` means that guessing interesting predicates is disabled. See 4.2.1.	`0`
`cpa.invariants .interestingVariableLimit`	The maximum number of variables to consider as interesting. `-1` one disables the limit, meaning that all variables are considered to be interesting, but this is not recommended. `0` means that guessing interesting variables is disabled. See 4.2.1.	`2`
`cpa.invariants.merge`	Which merge operator to use for InvariantCPA. Allowed values are `JOIN` and `SEP`. Using the option `SEP` turns off merging of states completely, rendering the options `cpa.invariants .interestingPredicatesLimit` and `cpa.invariants .interestingVariableLimit` irrelevant. Using the option `SEP` is not recommended.	`JOIN`
`cpa.invariants.useBitvectors`	Whether or not to use a bit-vector formula manager when extracting invariant approximations from states.	`false`

Table 4.1: Configuration options

Name	Description	Default
`cpa.loopstack.maxLoopIterations`	Threshold for unrolling program loops, which essentially maps to the k in k-induction as explained in 2.1. `0` means that loops are unrolled completely.	`0`
`cpa.predicate.solver` `.useIntegers`	Forces the satisfiability solver to use integer variables instead of floating point variables. Without this option, induction is often too imprecise to perform a successful proof.	`false`
`bmc.induction`	Use induction in the bounded model checking algorithm. Without using induction, bounded model checking does not apply the algorithm implemented in this thesis.	`false`
`bmc.useInvariantsForInduction`	Use invariants for the induction in the bounded model checking algorithm. Without this option, bounded model checking does not apply the algorithm implemented in this thesis for generating invariants to support the induction proof.	`false`

Table 4.2: Important external configuration options

4.4 SimpleInterval

SimpleIntervals are the basic building blocks for representing states in the InvariantsCPA. Their structure corresponds to the description of intervals given in 3.2.1.1. A SimpleInterval represents a contiguous range of values over $\mathbb{Z}$.

Such an interval may have a finite lower and a finite upper bound or be infinite in one or both directions. Single finite values are represented as an interval with two equal finite bounds.

An interval is never empty, i.e. it always contains at least one value, which is relevant when using intervals to represent knowledge about the state of a program value. SimpleIntervals are not suited to represent the state $\perp$ for a value. The unbounded interval, however, can be used to represent $\top$.

SimpleIntervals are immutable objects. All operations that return a modified version of the object they are invoked on either create new instances or reuse appropriate instances created earlier, but never modify the object itself. There are operations for intersecting, spanning over, negating or extending intervals as well as operations for checking whether or not a value is contained in an interval, two intervals touch or two intervals intersect each other. More complex operations, for example most arithmetic operations, are not provided by this class.

4.5 CompoundState

A CompoundState is a set of intervals that do not touch or intersect each other. CompoundStates are the preferred method of representing constant values in the InvariantsCPA. Like SimpleIntervals, CompountStates are immutable objects that cannot be changed over the public interface. Unlike SimpleIntervals, it is possible to represent non-consecutive values with CompoundStates or to use a CompoundState with an empty set of intervals to represent the state $\perp$. A CompoundState can also be seen as a compact approach to represent a possibly infinite set of integer values, so the set of CompoundStates C can be defined as $C := \mathcal{P}(\mathbb{Z})$ and for every CompoundState $c \in C$, $c \subseteq \mathbb{Z}$ is true. To represent their relation to SimpleIntervals, to shorten the notation of large states and to be able to express infinite sets of integer numbers, the notation used for CompoundStates in this thesis will be sets of intervals instead of sets of integer numbers unless a set of integer numbers is required for an explanation. CompoundStates match the description of lists of intervals in 3.2.1.1 and the example figures given in 3.1, 3.2, 3.3, 3.4 and 3.5;

CompoundStates provide all basic binary arithmetic and bit-wise operations used in the C programming language as well as the additive inverse and the binary negation over compound states. Some of those operations, like for example the additive inverse, are exact, while others, like for example the arithmetic multiplication, are less exact but still guarantee to produce states that are - not necessarily proper - supersets of the actual exact operation results. More formally, for two CompoundStates $c_1, c_2 \in C$ and a binary operator $o_c : C \times C \to C$ corresponding to a binary operator $o_i : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $\{o_i(i_{c_1}, i_{c_2}) | i_{c_1} \in c_1, i_{c_2} \in c_2\} \subseteq o_c(c_1, c_2)$. The same principle applies to the implemented unary operations: For a CompoundState $c \in C$ and an unary operator $o_c : C \to C$ corresponding to an unary operator $o_i : \mathbb{Z} \to \mathbb{Z}$, $\{o_i(i_c) | i_c \in c\} \subseteq o_c(c)$.

Allowing for inaccuracies within these bounds is valid, because the goal of the InvariantsCPA is not a C interpreter but to produce invariants within the context of program analysis. While higher precision leads to more useful invariants, striving for precision is only reasonable as long as it is feasible, and, as explained in 1.1, checking all possible concrete states of a program is generally considered to be infeasible even for finite state spaces.

To understand that the superset constraint defined above for the operations over CompoundStates ensures that while precision might decrease, correctness is preserved, consider the following example:

Let the CompoundState $s_v := \{2, 3, 4, 8, 9, 10\} = \{[2, 4], [8, 10]\}$ as shown in figure 3.4 be the model of a program value v and the CompoundState $s_c := \{2\} = \{[2, 2]\}$ be the model of a program value c. When modelling the result r of the multiplication $c * v$, without any further information the most exact model possible for r would be the set $\{4, 6, 8, 16, 18, 20\}$ or in CompoundState notation $\{[4, 4], [6, 6], [8, 8], [16, 16], [18, 18], [20, 20]\}$. It is obvious that multiplication with a constant factor would produce an interval with a single integer number for each integer number contained in the other operand. With the present implementation of CompoundStates, such exact models cannot efficiently be represented and are even impossible to represent when considering an operand modelling an infinite set of integer numbers. While technically the CompoundState implementation is able to represent sets containing many intervals, the number of intervals - not necessarily the number of values represented - is for reasons of efficiency kept small in practice. Therefore the actual multiplication operation for $s_v * s_c$ would yield a CompoundState $s_r := \{[4, 8], [16, 20]\}$. While numbers like 3, 7 or 19 are contained in this set but not in the exact result shown above, all correct values are contained as well. Therefore the model, which is meant to express that the actual program value at the program state represented by model is a value contained in the set, is still correct. Of course, special cases exist where an improved precision may be achieved for such operations with minimal loss of efficiency, but as they are subject to change regarding future tests and evaluations, those optimizations are not explicitly listed in this thesis.

4.6 InvariantsFormula

InvariantsFormulas represent the expressions handled by the InvariantsCPA. Their structure is very similar to the structure of the CExpression tree structure used by

CPAchecker, however, the latter is meant specifically to represent C expressions and thus does not allow other types of constants but CLiteralExpressions.

The generic implementation of the InvariantsFormula tree provides the InvariantsCPA with a type safe means to use expressions with different constant types, for example CompoundStates.

The visitor pattern has been implemented around InvariantsFormulas, which are the nodes accepting the visitors in terms of the pattern[30]. They are thus traversable by visitors. There is a classic implementation of the accept methods with just the visitors, as well as an extension with a generic parameter so that parameterized algorithms do not require the parameter to be bound to a visitor instance.

4.7 InvariantsState

Instances of the InvariantsState class are the abstract elements of the InvariantsCPA. They represent abstract program states determined during the analysis and are immutable objects.

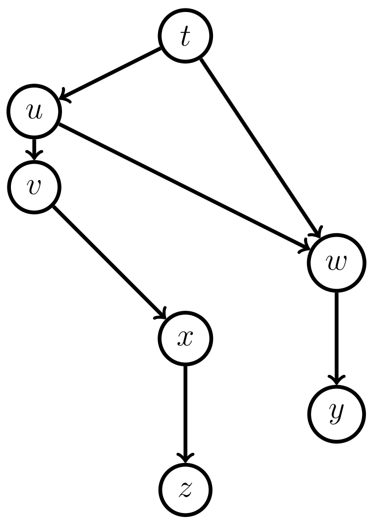
Two data structures are used to represent the state information: A mapping of expressions to variable names as the environment and a set of assumptions about binary interrelations between variables. All expressions used in InvariantsStates are InvariantsFormulas with CompundState constants.

The mapping is required to be able to quickly resolve variables contained in expressions. Resolving a variable in an expression means replacing the variable in the expression with the expression mapped to the variable. All variables that are not explicitly mapped to an expression are implicitly considered to be mapped to $\top$. Expressions mapped to variables may contain other variables. It is often desirable to apply variable resolution to an expression until no variables remain. This is only possible if there are no cyclic mappings. Therefore mappings like $\{x \to x\}$ or $\{x \to y, y \to x\}$ are forbidden. An example mapping and its representation as a directed acyclic graph are shown in figure 4.1;

The set of binary interrelations between variables provides the ability to represent some information that is simple yet not adequate for the mapping.

During the analysis, new information is gained and old information is invalidated. Changes in information happen by transitioning from one state over an edge to another one and by joining states from different paths. While the theoretical principles behind information extraction, information representation, and the joining of paths, are discussed in 2.2.1, 3.2.1 and 3.2.1.3, the following subsections 4.7.1, 4.7.2 and 4.7.3 explain the details of the implementation of these principles.

t	$u+w$
u	$v+w$
v	x
w	y
x	z
y	$\top$
z	$\top$



$$\begin{tikzcd}
 & t \arrow[dl] \arrow[ddrr] & & \\
u \arrow[d] \arrow[drrr] & & & \\
v \arrow[dr] & & & w \arrow[d] \\
 & x \arrow[d] & & y \\
 & z & &
\end{tikzcd}$$

Figure 4.1: A mapping of variables to expressions, representing knowledge about an environment

4.7.1 Assignment Transitions

The environment information is collected when the configurable program analysis hits an assignment edge in the control flow automaton. The InvariantsTransferRelation then calls the assignment function on the current state. This function does not change the immutable InvariantsState but rather returns a different instance reflecting the changed state.

When an array slot is assigned, sometimes special handling is required. If array subscripts were always constant, no special treatment would be required. However, array subscripts may contain references to other variables with possibly unknown values. If an array slot is assigned where the exact value of the subscript expression is unknown, all previously known information about any slots of the array that might be affected becomes invalid, and the assignment expression must not be stored as information about any specific slot of the array, but can only be stored in combination with the subscript. Conversely, if an array slot is assigned where the exact value of the subscript expression is known, all information stored about the array in combination with unknown subscript values becomes invalid and has to be removed. When the array itself is reassigned, this information must be removed, and when it is copied to another array variable, this information can be copied as well.

The more common uses of pointers can be dealt with easily. The handling of arrays has just been discussed already. Another common use of pointers often arises when objects or structs are used and their members are accessed via references. Assignments to those members can be stored just like assignments to other variables by simply using the complete dereferencing expression. The same can be done if there is no pointer involved and members are accessed directly by storing the information in combination with the access expression. When the object or struct is copied to another variable, this information must be copied accordingly, and when the object or struct itself is reassigned, the old information about its members must be dropped, similar to the way arrays are handled.

Pointers in general, however, pose a serious problem to the analysis, because they can be used in more complex ways than the previously described ones. Sometimes it is not known where a pointer points to. If such a pointer is assigned to, any previous information becomes invalid, because anything could have changed. Even if an absolute value of a pointer is known, there is usually no or not enough information about the memory layout of the program, so that it becomes impossible to determine the affected variables.

A similar problem arises from aliases. If a variable is assigned to that is an alias to a variable, the other variables is assigned to as well. If it is not known if the assigned variable is an alias to another variable, all information about variables that could be aliased by the assigned variable should be deleted. This is not yet implemented into the analysis, so there may be incorrect results for such cases. There is, however, an alias analysis implemented within CPAchecker that might be used in the future to resolve this issue.

The assignment function is not limited to modifying the environment. A reassignment of a variable might require the removal of assumptions from the set of assumptions described in 4.7.2 that are no longer valid.

Also, the assignment function may return the same state instance they were invoked on, provided that it is able to detect that the respective assignment transition does not change any information. It is also possible for the function to return `null`, if it detects that the resulting state model would otherwise represent $\bot$, for example when the assignment triggers a simplification of the environment that reveals a contradiction.

4.7.2 Assume Transitions

Whenever an expression is found at an assume edge, the information represented by the edge is combined with the previously known information to check whether any simple information about binary relations between variables, such as equalities or inequalities, can be gained. Any such relation found is stored in a set of assumptions separate from the environment.

Just like the assignment function is not limited to touching the environment, the assume function is not limited to modifying the assumption set. Environment information can be refined by combining it with information obtained from assume edges, but this refinement is only performed within bounds that do not violate the abstract interpretation strategy explained in 2.2.3: Only the first time the edge appears in the analyzed path or if the assumption expression contains no more than one variable, because this means that the contained variable is compared to a constant and will not produce different information the next time the edge is visited during the analysis of the path. This, of course, does not guarantee that non-`null` results of these functions represent reachable states, otherwise the analysis alone would be sufficient to prove a program correct, which is obviously the responsibility of the induction algorithm using the analysis, not the analysis itself.

Another likeness to the assignment function is that the assume function, too, may return the same state if the transition does not change any information or `null`, if it detects that the resulting state model would otherwise represent $\bot$, for example when assumptions in the assumption set contradict each other. Again, this does not guarantee that non-`null` results of these functions represent reachable states.

4.7.3 Joining States

As explained in 3.2.1.3, joining paths is required to ensure the convergence of the analysis, because while extracting information about a program in general seems like an easy task, the actual difficulty lies in deciding what information do drop and what information to retain over the course of the analysis so as not to explore a potentially infinite but certainly often large state space: Keeping every piece of information that might be required to prove correctness is generally considered as infeasible even for finite state spaces [5, p. 1]. When defining a configurable program analysis for CPAchecker, it is possible to specify the behavior of joining states. The contract of joining two states is to produce a state covering both input states by either precisely representing their united information or over-approximating it. As it is this over-approximation which actually causes the previously mentioned loss of information, it is in some cases desirable not to join states. Therefore it is also possible to refuse joining two states on a per case basis.

While the decision whether to join two states or not does not directly affect the correctness of the algorithm, it does affect the two opposing properties of precision and runtime. Joining two states might cause a loss of important information, while not joining enough states might cause the algorithm to take too much time. The decision is further complicated by not knowing in advance what information is important and not knowing in advance how much the runtime increases by not merging two specific state. To tackle at least one of those problems, 3.2.1.3 addresses strategies for guessing which variables might be important to the analysis. Whenever CPAchecker suggests merging two states to the analysis, it is checked if the states differ in important information. If they do, the states are not merged, otherwise they are. This way, information considered to be important is preserved, while information considered to be less important is over-approximated.

Consider for example the states $S_1 := \{y = \{[0, 0]\}, z = \{[0, 0]\}\}$ and $S_1 := \{y = \{(-Inf, -1], [1, Inf)\}, z = \{[1, 1]\}\}$. Merging the states would produce the state

$S_{1,2} := \{z = \{[0,1]\}\}$, because $(y = 0) \vee (y \neq 0)$ means that there is no information about y anymore. This example is actually the same as in figure 2.4 in the introduction to the theory of the invariant generation algorithm implemented in this thesis. The solution for this example was to keep the information apart by storing not the union of variables but rather the union of states. This means we would have to represent the information $((y = 0) \wedge (z = 0)) \vee ((y \neq 0) \wedge (z = 1))$. While this cannot be represented by a single InvariantsState, this is exactly the information represented by not merging the two states. At the same time, the example shows why merging states in general is desirable: Not merging two states obviously has about at least twice the memory footprint as merging them, and as the analysis has to take transitions from all states into account, every time two states are merged, one state less must be analyzed.

In the same way that environment information can be used for discrimination between pairs of states that should be merged and pairs of states that should not be merged, the discrimination can also be based on generic predicates. This feature, however, is turned off by default.

4.7.4 Invariant Extraction

The most important operation provided by InvariantsStates is the extraction of invariants in a format recognized by applications using the algorithm. This operation is implemented as a function that creates an SMT formula combining all assumptions extractable from the state it is invoked on, including all environment information, with conjunctions. This boolean formula is not an InvariantsFormula, but a formula created by a formula manager provided by the caller. A disjunction of the boolean formulas extracted from all states of a certain location produces an invariant for that location.

4.8 InvariantsTransferRelation

The InvariantsTransferRelation class defines the transfer relation of the implemented configurable program analysis using the singleton pattern[30]. The single instance of this class is thus responsible for computing the successors reachable from a given InvariantsState over a given control flow edge.

The InvariantsTransferRelation will produce either no successor or one successor for any given edge or predecessor, but never more than one. Edges representing assumptions will generally add information, edges representing assignments might also remove information. The successive InvariantsState represents the information of the predecessor modified to reflect the change in information represented by the transition over the edge. The successor may be equal to the predecessor if the same information is represented before and after the transition. If the new information reveals an obvious contradiction, no successor is produced.

The actual implementation of the creation of successors is part of the InvariantsState class described in 4.7, respectively its functions for computing the successor of a state over an assignment edge or over an assume edge.

4.9 Evaluation of Formulas

As explained in 3.2.2, this thesis uses two different strategies for expression evaluation. One of the strategies is used to evaluate any given expression as exact as possible. This strategy is implemented by a formula visitor. The other strategy is used to first evaluate any given expression as exact as possible, like the first strategy, but then abstract it in the way described in 3.2.2. This strategy is implemented by a formula visitor as well.

5 Experimental Results

This chapter will show the experimental results of this thesis. The experiments made were very extensive, resulting in a large amount of data. To improve readability, only excerpts are shown in this part of the document, while the complete data can be found on the CD accompanying this thesis.

5.1 Selected Benchmarks

To be able to properly evaluate the implemented algorithm, all of the programs used for the benchmarks were chosen from the set of benchmarks that was also used for the second International Competition on Software Verification of 2013[1]. The complete set of benchmarks used for the experiments conducted for thesis therefore contains more than two thousand different programs.

A small subset of this set was selected for more specific experiments. This selection consists of a large part of the simplified and the normal `ssh` programs from the competition benchmark set, because while reasonably complex, these programs also clearly expose the property of containing a loop where the body is split in different cases, that was described as a potential indicator for programs that benefit from k-induction in 3.1. Moreover, each of the programs contains only one single loop, which is a requirement for the k-induction strategy implemented in CPAchecker's bounded model checking algorithm.

5.2 Analyses

5.2.1 Bounded Model Checking

The primary motivation for the implementation of this light-weight approach to invariant generation is the k-induction strategy implemented in CPAchecker's bounded model checking algorithm. Therefore the only reasonable way of determining the viability of the approach for the purpose it was designed for is by applying the bounded model checking analysis to adequate programs.

[1]`http://sv-comp.sosy-lab.org/2013/`– last check: October 10, 2013

5.2.2 Predicate Analysis

The k-induction strategy implemented for bounded model checking within CPAchecker is only applicable to programs with just one single loop and is only used for programs where no error is found for unrolling k iterations of the loop, so the invariant generation algorithm is usually not applied for unsafe programs. Using bounded model checking is therefore not the adequate option for benchmarking the invariant generation algorithm in general over the whole set of benchmarks described in 5.1, because many of the contained programs contain multiple loops or are unsafe. Fortunately, CPAchecker's predicate analysis is also able to use the algorithm to extract invariants for the analyzed programs, so this analysis was used to conduct experiments over the large set of benchmarks.

5.3 Measured Properties

Alongside the conducted experiments, several properties were measured.

5.3.1 Safety

For all of the programs used for the experiments, it is known in advance whether the program is safe or unsafe. As a result can be either `SAFE`, `UNSAFE` or `UNKNOWN`, each experiment might therefore either determine safety or unsafety correctly, result in a false positive or false negative, or make no conclusive statement about the safety of the program. To be able to save space to present comparisons between different configurations without the reader having to jump between pages, `SAFE`, `UNSAFE` or `UNKNOWN` will be shortened to `S`, `U` and `?` within the tables.

It is also possible for an experiment to be terminated prematurely if too much time or memory is consumed, or an exception might occur during the execution. While these cases basically count as `UNKNOWN` results as well, they are obviously much less desirable than the algorithm coming to the conclusion that it is unable to determine the analyzed program's safety on its own accord.

It should be noted that for the bounded model checking algorithm to determine the unsafety of a program, the invariant generation algorithm implemented in this thesis is not applied. Correct results for programs known to be unsafe are thus much less relevant for the evaluation of the presented algorithm than the results of programs known to be safe, as k-induction is applied to prove safety. An experiment were a program containing a loop is correctly verified as `SAFE` by bounded model checking shows that the generated invariants were sufficient for the k-induction proof to succeed with the given value of k, while an `UNKNOWN` result means that either the precision of the generated invariants or the value of k used were too low.

5.3.2 Time and Space

The execution time and the amount of memory required to run an experiment are important properties: Not only do they represent limited resources that are used to cancel experiments that use excessive amounts of them. There is also the conflict between time and space efficiency on the one hand and precision on the other hand, that continually emerged during the discussion of various aspects of this thesis in the previous chapters. Measured time values are given in seconds and displayed rounded to the next decade.

It is expected that higher memory consumption roughly correlates with higher CPU time and in contrast, that lower memory consumption correlates with a lower CPU time, because both properties are influenced by the common factor of the amount of states used by the invariant generation analysis. More states obviously lead to a higher memory consumption, but also require more processing capacities. Memory values are given in MB and rounded to the next hundred's place.

5.4 Benchmark Environment

The measurements were run on Intel Core i7-2600K machines with eight cores and 16 to 32 gigabytes of memory each. Each measurement was limited to a runtime of 15 minutes, a memory consumption of 15000 megabytes, a heap size limit of 13000 megabytes and 4 CPU cores. The CPAchecker version used was 1.2, revision 8828.

5.5 Experiments

The following subsections describe the conducted experiments in detail.

5.5.1 Proof of Concept

It has already been explained that the main motivation for this thesis is the application of k-induction. Table 5.2 shows the application of bounded model checking with k-induction to the `ssh` program selection described in 5.1. The configuration options used are shown in table 5.1.

As the generated invariants are used to prove loop safety in bounded model checking with induction, it is expected that the invariant generation has no impact on proving programs to be unsafe. Furthermore, it is expected that higher values of k allow for proving more programs to be safe by k-induction or unsafe by finding a counter example within k loop iterations, than with lower values of k. Because of the properties of k-induction discussed in 2.1, it should never happen that program safety is unknown for a certain value of k, when it was proven to be safe or unsafe with a lower value of k.

To be able to show the differences in the ability to verify the programs depending on the value of k, only the verification results are shown. With only one exception, all safe programs are verified for $k \geq 4$, while lower values of k are insufficient for the verification of some programs. The table also shows that any program verified for a certain value v for k is also verified for every $k > v$, as is expected and required because of the properties of k-induction described in 2.1. This set of experiments can thus be seen as a kind of proof of concept for k-induction in combination with the generated invariants.

Looking at figure 5.2, the consumed total CPU time for each value of k seems to be inversely proportional to the value of k. This can be explained by the fact that the bounded model checking algorithm first checks for an error path of length k and only invokes invariant generation to attempt an induction proof for safety if no such path is found. Therefore, the more unsafe programs are correctly found to be unsafe by the algorithm, the fewer invocations of invariant generations occur, thus decreasing the overall time consumption. As soon as increasing k does not reveal any further error paths, no more time can be saved via this phenomenon, and time consumption is actually expected to rise due to the cost of unrolling the loop iterations. Indeed, the total consumed CPU time for $k = 3$ is higher in the displayed experiments than for $k = 2$, while for both values of k, the programs that are proven to be unsafe by the analysis are the same. In contrast, when increasing k from $k = 5$ where 11 programs are known to be unsafe to $k = 6$, where 19 programs are known to be unsafe, total CPU time required to process all files is about cut in half. As expected, apart from some minor fluctuations, memory consumption is high when CPU times are high and low, when CPU times are low. The memory chart is shown in figure 5.3.

The programs where safety is still `UNKNOWN` for $k = 4$ are, with only one exception, actually unsafe as is shown by continuing to increase k until $k = 10$. The verification result chart is shown in figure 5.1.

To determine the actual benefit of the invariant generation algorithm, these results have to be compared with another run of the experiments where the usage of the invariants is turned off, so they were repeated with the configuration options shown in table 5.1, with the difference of `bmc.useInvariantsForInduction` being set to `false`. It is expected that the total CPU time consumed is lower than when using invariant generation, and that it increases proportionally with k because of the cost of unrolling the loop iterations.

Table 5.3 shows that without invariant generation, far less programs can be proven to be safe, and those that can be proven mostly require a higher value of k. The unsafety of programs, on the other hand, is proven for the same values of k as in the earlier set of experiments shown in table 5.2. So while k-induction obviously still is able to improve the results for higher values of k, for example by proving ssh-simplified/s3_srvr_1a_safe

Name	Value
`cpa.invariants.analyzeTargetPathsOnly`	`true`
`cpa.invariants.analyzeRelevantVariablesOnly`	`true`
`cpa.invariants.interestingPredicatesLimit`	`0`
`cpa.invariants.interestingVariableLimit`	`2`
`cpa.invariants.merge`	`JOIN`
`cpa.invariants.useBitvectors`	`false`
`cpa.loopstack.maxLoopIterations`	`1-10`
`cpa.predicate.solver.useIntegers`	`true`
`bmc.induction`	`true`
`bmc.useInvariantsForInduction`	`true`

Table 5.1: Configuration options used for the bounded model checking experiments on the `ssh` program selection

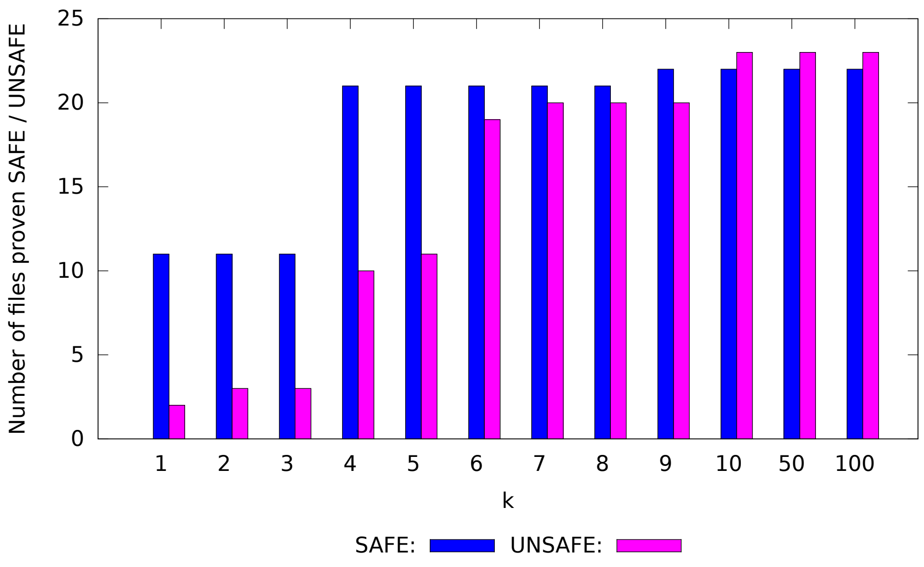


Figure 5.1: The verification result chart for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set. In total there are 22 safe and 23 unsafe programs in the set.

File	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
s3_srvr_10_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_11_unsafe	?	?	?	?	?	?	**U**	**U**	**U**	**U**
s3_srvr_12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr_13_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_14_unsafe	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_1_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1a_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_1b_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_2_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_2_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_3_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_4_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_6_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_6_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_7_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_8_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.01_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.01_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.02_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.02_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.03_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.04_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.06_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.06_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.07_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.07_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.08_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.08_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.09_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.09_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.10_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.10_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.11_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr.blast.11_unsafe	?	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.12_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.13_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.13_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.14_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.14_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.15_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.15_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.16_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.16_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
CPU time in s	2620	2300	2340	2260	2240	1000	1000	1020	1040	1020
Total memory in MB	72000	67000	65300	61700	58800	43500	44400	45000	42600	41700
Safe results	11	11	11	21	21	21	21	21	22	22
Unsafe results	2	3	3	10	11	19	20	20	20	23

Table 5.2: Bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set

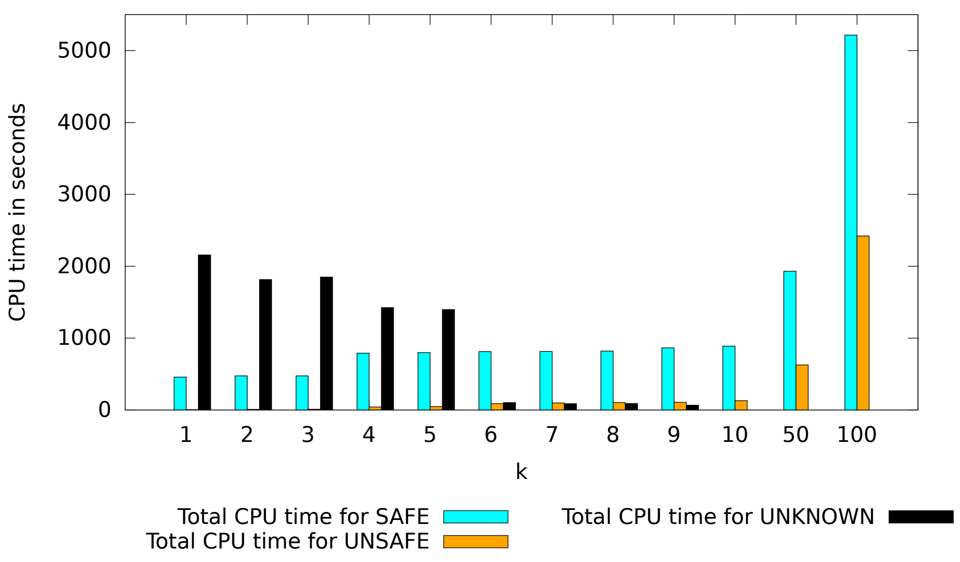


Figure 5.2: The consumed CPU time graph for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set

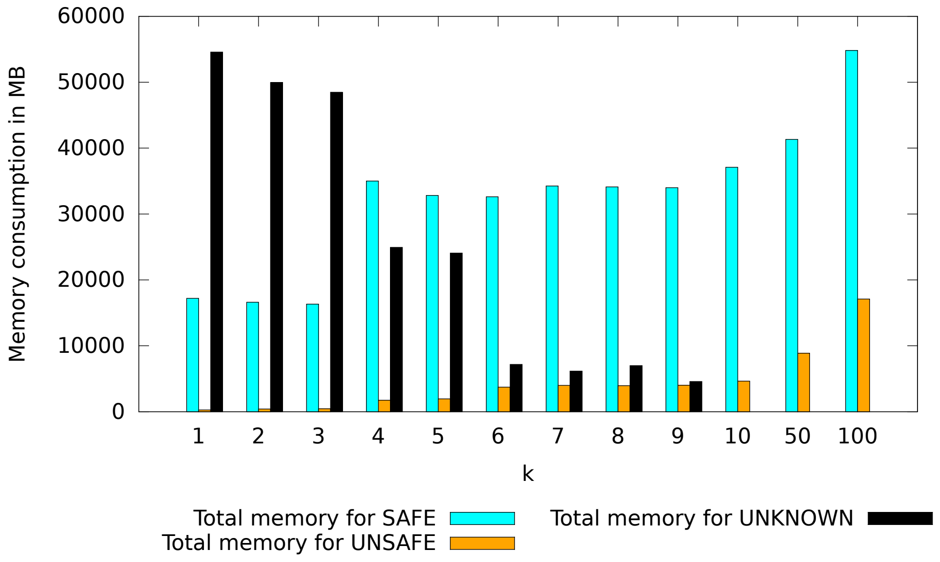


Figure 5.3: The consumed memory graph for bounded model checking with k-induction using invariant generation applied on the `ssh` benchmark set

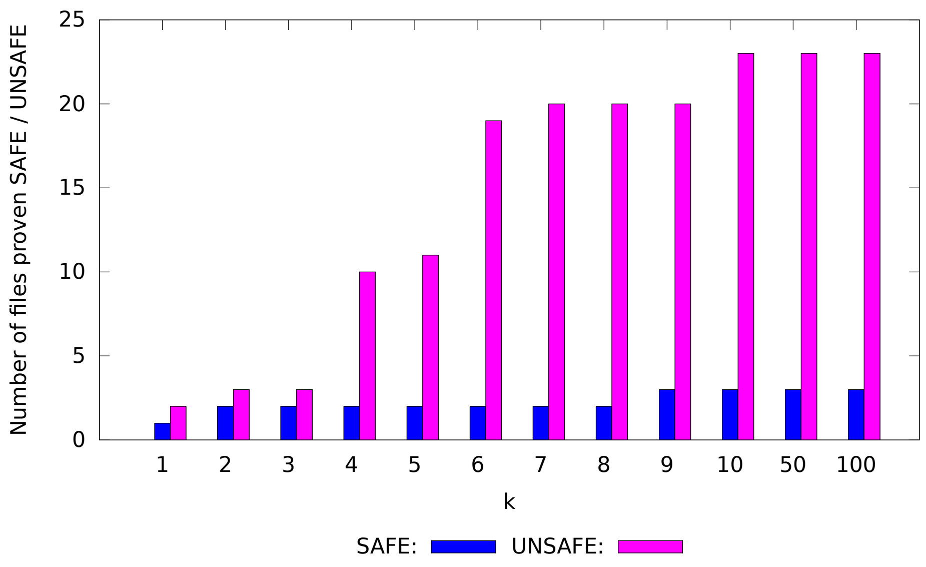


Figure 5.4: The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation. In total the set contains 22 safe and 23 unsafe programs.

to be safe for $k = 9$ when its safety is still unknown for $k = 8$. The verification result chart is displayed in figure 5.4.

Memory consumption, as shown in figure 5.6, seems to relate to CPU time consumption even more strongly than with invariant generation. Looking at the CPU time chart in figure 5.5, the total consumed CPU time is a lot lower than when invariant generation is enabled and, as expected, the more loop iterations are unrolled, the higher the consumed CPU time. Seeing the unsatisfying overall results, however, shows how important the generated invariants are for the induction proofs. While not shown in the table, even using $k = 100$ does not improve the results of the experiments any further.

5.5.2 Guessing of Potentially Interesting Variables

As discussed in 3.2.1.3 and 4.2.1, a strategy for selecting certain variables as more interesting to the analysis than others is employed to decide when to merge states and when not to. Because the usage of this strategy is optional, it is easy to test how it affects the outcome of the verification by repeating the experiments of 5.5.1 with the configuration options shown in table 5.1 with the change that `cpa.invariants .interestingVariableLimit` is set to `0` or to `1`.

As it turns out, not using any selection of variables to prevent merging of states, thus merging always, is no better than not using the invariant generation algorithm at all.

File	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
s3_srvr_10_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_11_unsafe	?	?	?	?	?	?	**U**	**U**	**U**	**U**
s3_srvr_12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr_13_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_14_unsafe	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_1_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1a_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr_1b_safe	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_2_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_2_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_3_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_4_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_6_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_6_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_7_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_8_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.01_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.01_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.02_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.02_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.03_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.04_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.06_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.06_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.07_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.07_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.08_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.08_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.09_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.09_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.10_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.10_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.11_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.11_unsafe	?	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.12_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.13_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.13_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.14_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.14_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.15_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.15_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.16_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.16_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
CPU time in s	160	180	200	210	220	230	240	250	280	300
Total memory in MB	6500	7400	8000	8100	8500	8400	8800	8800	9100	9400
Safe results	1	2	2	2	2	2	2	2	3	3
Unsafe results	2	3	3	10	11	19	20	20	20	23

Table 5.3: Bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation

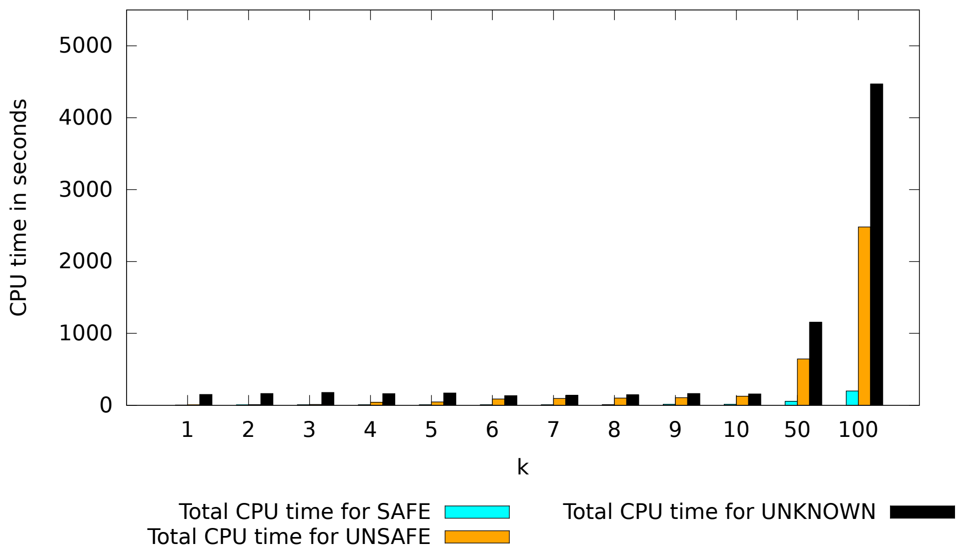


Figure 5.5: The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation

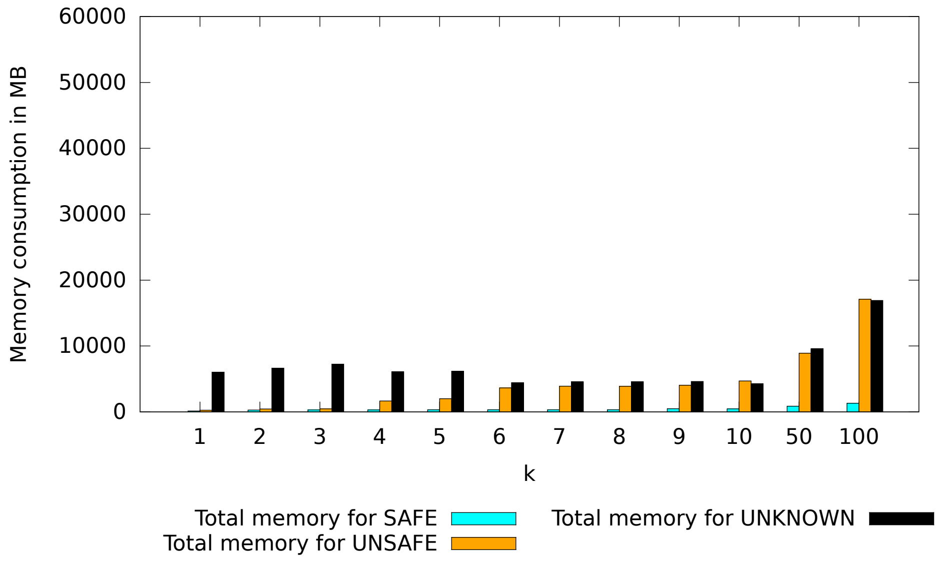


Figure 5.6: The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, not using invariant generation

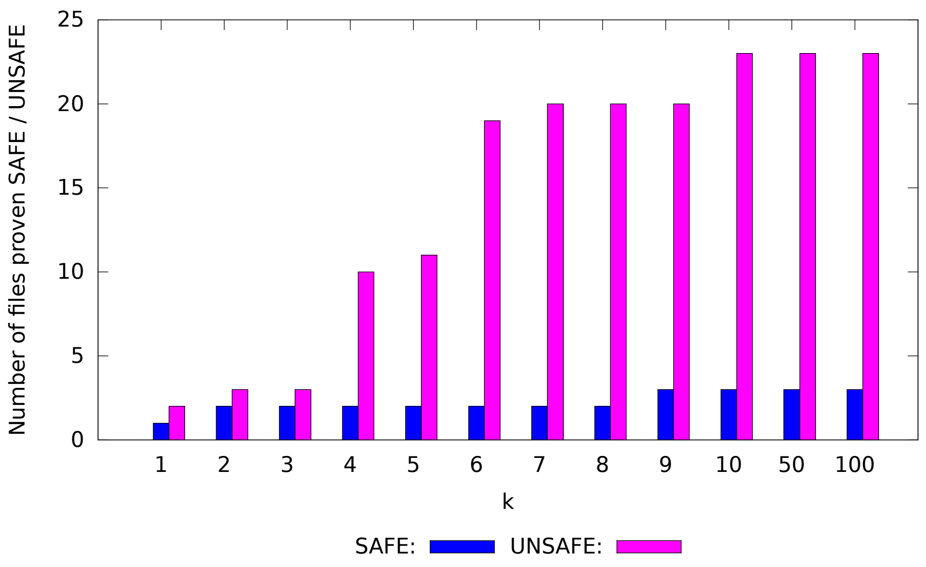


Figure 5.7: The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing one important variables. In total the set contains 22 safe and 23 unsafe programs.

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
CPU time in s	390	390	410	410	420	380	380	400	430	430
Total memory in MB	12800	12400	12300	12100	12400	11400	11700	11800	12400	11900
Safe results	1	2	2	2	2	2	2	2	3	3
Unsafe results	2	3	3	10	11	19	20	20	20	23

Table 5.4: CPU times and memory consumption for the different values of k when using bounded model checking with k-induction and invariant generation, setting `cpa.invariants.interestingVariableLimit` to `0`

The verification results are the same as in table 5.3. This emphasizes the importance of making the right decision about when to merge states. The total CPU time used for each value of k is shown in table 5.4. Similar to the the experiments where invariant generation was switched off shown in table 5.3, CPU time increases when k increases. Due to the overhead of the invariant generation algorithm, overall CPU time is higher compared to not using invariant generation, but being able to always merge states seems to keep this overhead low enough so that the direct proportionality between k and the CPU time is exposed, even though higher values of k allow for skipping invariant generation for unsafe programs. Only the notable gap between $k = 5$ and $k = 6$ is an exception to this rule. A similar effect can be observed on memory consumption.

Setting `cpa.invariants.interestingVariableLimit` to `1`, as shown in table 5.5 and figure 5.10, at least allows the verification of some of the programs, albeit partially

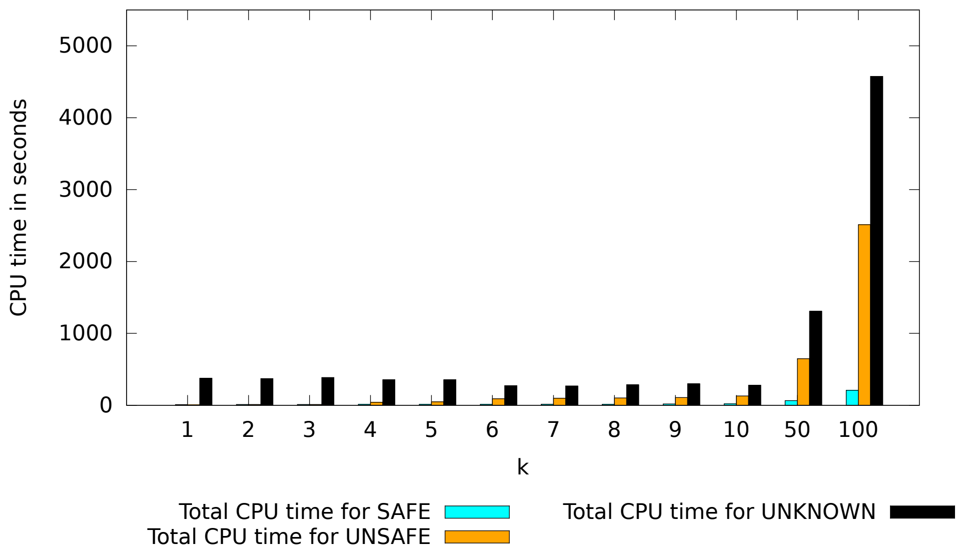


Figure 5.8: The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing important variables

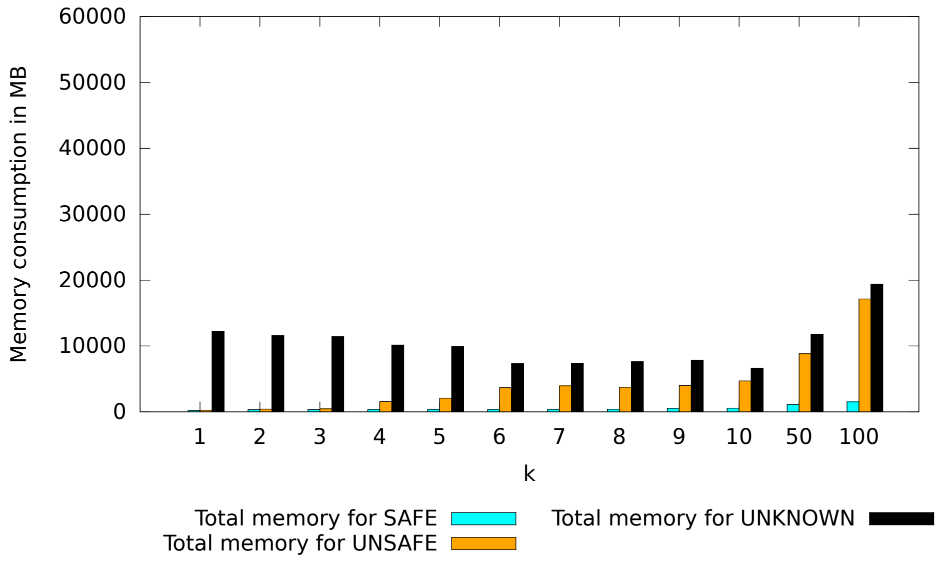


Figure 5.9: The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation but not guessing important variables

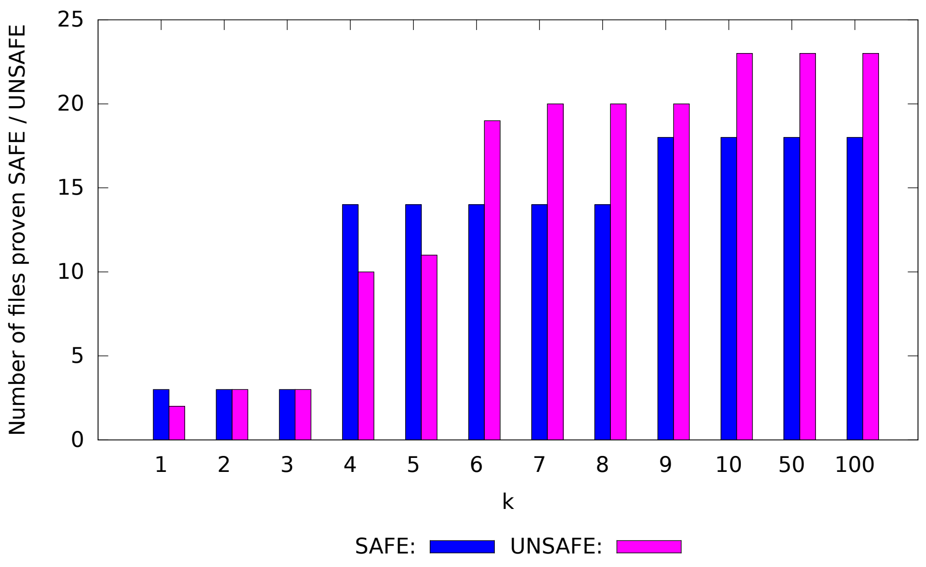


Figure 5.10: The verification result chart for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable. In total the set contains 22 safe and 23 unsafe programs.

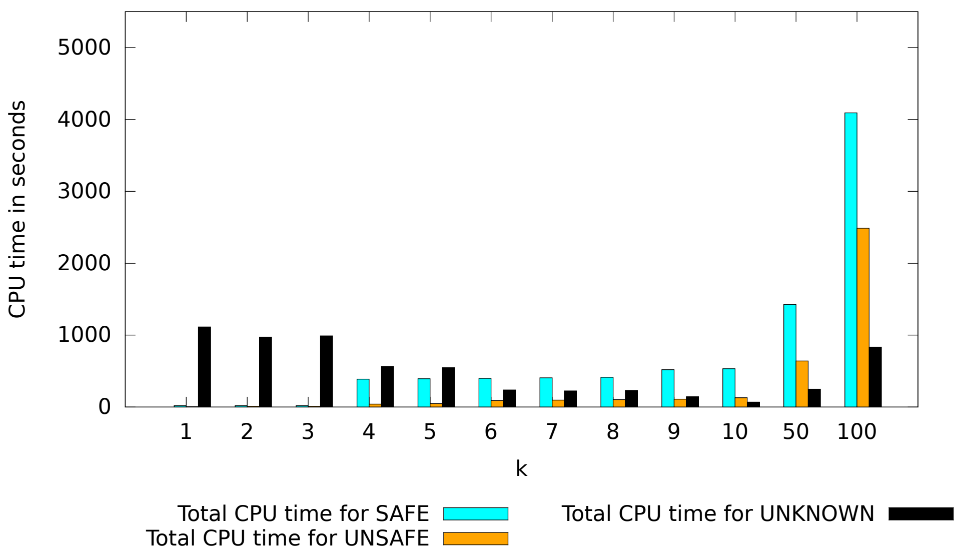


Figure 5.11: The consumed CPU time graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable

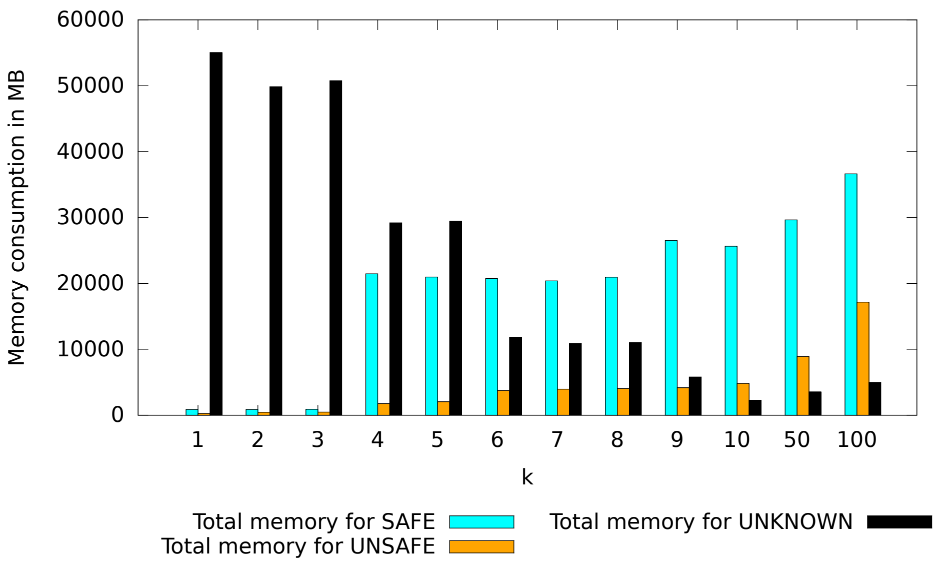


Figure 5.12: The consumed memory graph for bounded model checking with k-induction applied on the `ssh` benchmark set, using invariant generation with guessing one important variable

higher values of k are required than when using the default value of `1` for the limit. Also, for some safe programs that can be verified without the invariant generation or with `cpa.invariants.interestingVariableLimit` to `0`, lower values of k are sufficient to verify safe programs when using invariant generation with `cpa.invariants .interestingVariableLimit` set to `1`. This observation strongly supports the idea that higher values of k enable induction to succeed with weaker invariants and that inversely, using induction lower values of k requires stronger invariants.

With one variable being considered as interesting, the resulting fewer state merges and thus the higher amount of states are reflected by the total CPU times and memory consumption values, which are a lot higher than for always merging or for not using invariant generation at all. At the same time, the CPU times and memory consumption are lower than for using a limit of 2 as can be seen by comparing the figures 5.11 and 5.12 with the figures 5.2 and 5.3.

Again, although not shown in the tables, the runs were also executed with $k = 100$ without improving the results of the experiments any further.

5.5.3 Comparative Benchmark

As explained in 5.2.2, bounded model checking experiments alone are not sufficient to show the practicability of the invariant checking algorithm. Therefore, the large benchmark set mentioned in 5.1 was analyzed with three different configurations:

File	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
s3_srvr_10_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_11_unsafe	?	?	?	?	?	?	**U**	**U**	**U**	**U**
s3_srvr_12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr_13_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_14_unsafe	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_1_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_1a_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_1b_safe	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_2_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_2_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_3_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_4_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr_6_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr_6_unsafe	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr_7_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr_8_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.01_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.01_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.02_safe	?	?	?	?	?	?	?	?	?	?
s3_srvr.blast.02_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.03_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.04_unsafe	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.06_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.06_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.07_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr.blast.07_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.08_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.08_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.09_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.09_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.10_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.10_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.11_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr.blast.11_unsafe	?	?	?	?	**U**	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.12_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.12_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.13_safe	?	?	?	?	?	?	?	?	**S**	**S**
s3_srvr.blast.13_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.14_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.14_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
s3_srvr.blast.15_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.15_unsafe	?	?	?	?	?	?	?	?	?	**U**
s3_srvr.blast.16_safe	?	?	?	**S**	**S**	**S**	**S**	**S**	**S**	**S**
s3_srvr.blast.16_unsafe	?	?	?	?	?	**U**	**U**	**U**	**U**	**U**
CPU time in s	1140	1000	1020	1000	990	730	730	750	770	730
Total memory in MB	56200	51200	52100	52400	52400	36300	35200	36000	36500	32700
Safe results	3	3	3	14	14	14	14	14	18	18
Unsafe results	2	3	3	10	11	19	20	20	20	23

Table 5.5: Bounded model checking with k-induction applied on the `ssh` benchmark set, using `cpa.invariants.interestingVariableLimit = 1`

Analysis	`-predicateAnalysis -PredAbsRefiner-ABEl`	`-predicateAnalysis -invariants`	`-bmc-induction`
CPU time in s	160900	357400	823900
Total memory in MB	1312100	2397800	6068500
Correct results	1989	1491	865
Unknown results	0	0	355
False negatives	5	8	14
False positives	89	52	53
Safe results	1555	1425	535
Unsafe results	528	126	397
Timeouts	125	323	815
Memory exceedings	2	2	80
Crashes	76	410	104

Table 5.6: Comparison between three different analyses over the large benchmark set

Predicate analysis without support by the invariant generation algorithm, predicate analysis with invariant generation, and bounded model checking with induction using invariant generation. The most important aspect the results of the analyses need to be examined for is whether invariant generation leads to incorrect results where other strategies succeed, exceeds time or memory limits where other strategies do not, or even crashes when other strategies terminate normally, because these results show errors or deficits in the invariant generation algorithm or its integration with predicate analysis. As the invariant generation was designed with k-induction in mind, overall verification success is not expected to improve much over plain predicate analysis and might even be worse in some cases. Further aspects that can be observed are other differences in consumption of time or space, as well as different verification results in general.

Predicate analysis without invariant generation uses the configuration setting `-predicateAnalysis-PredAbsRefiner-ABEl`, predicate analysis with invariant generation uses the configuration setting `-predicateAnalysis-invariants`, and bounded model checking with induction and invariants uses the `-bmc-induction` configuration setting with $k = 10$. All three strategies use the default invariant generation specific parameters stated in 4.1 and 4.2.

Table 5.6 shows a summary of the results. The first and most obvious observation that can be made is that predicate analysis without invariant generation produces far more correct results, is faster and uses less memory than predicate analysis with invariant generation. Moreover, there are even a few more false negatives with invariant generation than without. The only immediately visible aspect where predicate analysis with invariant generation beats predicate analysis without invariant generation is the lower number of false positives. At first, this seems as if invariant generation mostly worsens the analysis. A closer view reveals a different view on the results, however.

One of the three false negatives produced with invariant generation but not without is the result of an experiment that exceeds the time limit without invariant generation, meaning that the false negative might also appear without invariant generation and was simply covered by the timeout. The other two false negatives are actual errors in invariant generation, likely occurring due to the lack of pointer alias handling discussed in 4.7.1.

More important is finding the causes for the large difference in correct verification results as well as the lower rate of false positives. Many cases where predicate analysis without invariant generation produces correct results or false positives when the same analysis with invariant generation does neither are due to program crashes or timeouts.

Debugging the crashes shows that the program crashes actually do not occur during the invariant generation but when predicate analysis uses the invariants and variables occur that are unknown to the analysis, for example because they are not in scope at a specific location. While such an invariant certainly is of no use to the verification, it should simply be ignored rather than leading to a program crash. Other solutions might be including a scope analysis into invariant generation or requesting information about specified variables only when extracting the invariants from the states produced by the invariant generation.

Debugging the timeouts reveals that the invariant generation terminates successfully, but that the SMT solver employed by the analysis is unable to solve the formulas produced by enhancing the information collected by the standard analysis with generated invariants. While intuitive expectations might suggest that giving additional information to the SMT solver by providing it with invariant assertions reduces the difficulty of the task, satisfiability is an NP-complete task[31] and cases where increasing the size of the checked formulas result in higher runtimes must be expected.

It is interesting to observe that while the numbers of programs verified by the two predicate analyses to be safe are fairly close to each other, while their numbers of programs proven unsafe are vastly different, and that even though there are a lot of SMT solver timeouts for predicate analysis with invariant generation, there are also many programs that can be verified with invariant generation but exceed the time limit without invariant generation.

Overall, these experiments can be seen as a success, as the occurring errors are integration problems, not problems caused specifically by the invariant generation algorithm, and the occurring timeouts are likely caused by the increased complexity of the constructed formulas presented to the SMT solver after adding the generated invariants. The results also show the importance of determining which classes of programs are likely to benefit most from verification with k-induction, a question already raised in 3.1.

5.5.4 Related Benchmarks

When Donaldson et al. employed k-induction to loop verification presenting their work about combined-case k-induction, they also provided a set of benchmarks and the experimental results they received from running their implementation over those benchmarks. The invariant generation algorithm implemented in this thesis is not applicable to those benchmarks as of yet, because the k-induction used by bounded model checking in CPAchecker currently only handles single loop programs and the benchmark set used by Donaldson et al. contains many multi-loop programs. Furthermore, Donaldson et al. use their verification tool to check for direct memory access races, a check that is not implemented in CPAchecker. No meaningful comparisons can therefore be made between their work and this thesis. The only potentially interesting observation is that their required values of k are between 1 and 4, just like in the benchmarks shown in 5.5.1. Due to the previously mentioned incomparability of the different benchmarks, the significance of this observation, however, is questionable.

6 Conclusion

This chapter concludes the thesis by giving a summary of the discussed aspects and properties of the implemented light-weight approach to invariant generation for software verification.

6.1 Summary

The first chapter gave an introduction to software verification in general and the idea of using k-induction with bounded model checking in particular, before proceeding to discuss related research and discussing some of the terminology and conventions used in this thesis.

The second chapter provided information about the theoretical background required to understand the ideas and strategies applied over the course of this elaboration. k-induction was introduced, the basics of invariant generation were explained and the definition of a configurable program analysis was given.

The third chapter discussed the concrete theory the implemented invariant generation is based on. It showed how invariant assertions are used to support k-induction, what techniques are used in this thesis to generate invariant assertions and the properties of a configurable program analysis specifically designed to implement the invariant generation algorithm were defined.

The fourth chapter elaborated on the details of the implementation of the algorithm within the CPAchecker framework. The important classes representing the components of the configurable program analysis were discussed and configuration and initialization of the `CPA` were explained.

The fifth chapter presented the evaluation results and showed that there are many programs that can successfully be proved by automatic software verification using bounded model checking and k-induction, and that there is even one partial benchmark subset the technique works particularly well on. It also showed, however, that there are many cases this strategy is not suitable for and that there is still work to be done on the integration of the invariant generation algorithm into the surrounding framework using it.

This sixth and last chapter summarizes the thesis and discusses aspects which must still be improved.

6.2 Prospects

The first steps into using k-induction to support automatic software verification with CPAchecker made in this thesis showed promising results. However, there is still work to be done and there are improvements to be made.

6.2.1 Integration into the existing framework

While the integration of the invariant generation into bounded model checking seems to work very well already, the same thing can not yet be said about its integration into predicate analysis. If the goal of using the invariant generation for other means than supporting k-induction is pursued, these issues need to be addressed.

The main issue was a program crash occurring because predicate analysis assumes that it knows every variable that the invariant generation algorithm might provide information about. The invariant generation, however, uses some helper variables to represent unknown array slots or function return values. Furthermore, the algorithm does not take variable scoping into account. The problem arising from the use of helper variables has already been fixed in a recent revision, but the general problem remains. One possibility to solve this problem might be including a scope analysis into invariant generation, another option would be requesting information about specified variables only when extracting the invariants from the states produced by the invariant generation. While the latter option is probably easier to implement, the former might bring additional benefits, because information about variable scoping can be used to drop information that became irrelevant after the concerned variables went out of scope.

Another issue observed were the timeouts occurring. It is generally hard to predict how much time it takes an SMT solver to solve a given formula. While the assumption that larger formulas result in longer solving times is backed by some results, counterexamples to this theory are also easy to find in the benchmarks. Adding more meaningful configuration options to the `CPA` might allow the analyses using invariant generation to to more fine-tuning. Perhaps this is sufficient to help preventing future timeouts. Otherwise, the invariant generation algorithm itself might have to be optimized.

6.2.2 Multi-Loop Programs

Currently, the k-induction algorithm implemented in the bounded model checking strategy of CPAchecker is unable to handle programs with more than one loop. This is an issue that often prevents the usage of invariant generation at all. Donaldson et al. already showed multiple approaches to solve these problems. They first presented split-case k-induction where all loops of a program are transformed into one monolithic loop which then can be verified with k-induction [27, p. 21 f.]. Then, they discussed combined-case k-induction, where they elaborated on cutting loops one at a time[8]. Both approaches can be considered to solve the multi-loop problem.

6.2.3 Bit-Vectors

It has been mentioned that the implemented invariant generation algorithm assumes that all values it deals with are arbitrary integers from an infinite range. In reality, integer values used in programs are actually bit-vectors with limited ranges. Stepping over one border then results in entering the range from the other side, a phenomenon also known as arithmetic overflow. When this phenomenon is encountered, the invariant generation produces incorrect results.

On the one hand, solving this problem seems complicated, as the currently used form of representing values does not easily translate to the different concept. On the other hand, the invariant generation program analysis was designed so that the value representation type is in theory interchangeable. Of course, many program parts do rely on operations specifically provided by the current representation due to reasons of efficiency which were the initial argument for using this specific implementation.

6.2.4 Pointer Aliasing

Currently, the invariant generation `CPA` ignores the possibility of pointer aliases, which potentially is the reason for some of the incorrect results observed in the benchmarks. This issue could be resolved by providing the analysis with information about pointer aliases. As CPAchecker already contains a pointer alias analysis, this task seems feasible.

6.2.5 Further Heuristics

A heuristic for guessing which variables might be especially interesting to the analysis has been implemented and the experiments showed that this heuristic is a significant factor for the quality of the invariant generation algorithm. However, it also affects the execution time and memory consumption negatively. It is desirable to explore further options for heuristics to improve the relation between precision and resource consumption and to evaluate these options against the results obtained in this thesis. Potential for such an option has already been discussed in the last part of 3.2.2, where the possibility of smartly increasing the precision of the abstraction strategy was mentioned.

Bibliography

[1] T. Hoare, "The Verifying Compiler: A Grand Challenge for Computing Research," *J. ACM*, vol. 50, pp. 63–69, Jan. 2003.

[2] A. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *London Mathematical Society*, vol. s2-42, pp. 230–265, november 1936.

[3] P. Bjesse, "What is Formal Verification?," *SIGDA Newsl.*, vol. 35, Dec. 2005.

[4] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear Invariant Generation using Non-linear Constraint Solving," in *In Computer Aided Verification*, pp. 420–432, Springer Verlag, 2003.

[5] R. Jhala and R. Majumdar, "Software Model Checking," *ACM Comput. Surv.*, vol. 41, pp. 21:1–21:54, Oct. 2009.

[6] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model Checking: Algorithmic Verification and Debugging," *Commun. ACM*, vol. 52, pp. 74–84, Nov. 2009.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[8] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software Verification using k-Induction," in *Proceedings of the 18th international conference on Static analysis*, SAS'11, (Berlin, Heidelberg), pp. 351–368, Springer-Verlag, 2011.

[9] D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 184–190, Springer Berlin Heidelberg, 2011.

[10] R. W. Floyd, "Assigning Meanings to Programs," in *Proceedings of a Symposium on Applied Mathematics* (J. T. Schwartz, ed.), vol. 19 of *Mathematical Aspects of Computer Science*, (Providence), pp. 19–31, American Mathematical Society, 1967.

[11] R. W. Floyd, "The Verifying Compiler," annual report, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 1967.

[12] J. C. King, *A Program Verifier.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 1970. AAI7018026.

[13] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[14] B. Wegbreit, "The Synthesis of Loop Predicates," *Commun. ACM*, vol. 17, pp. 102–113, Feb. 1974.

[15] M. Caplain, "Finding Invariant Assertions for Proving Programs," in *Proceedings of the international conference on Reliable software*, (New York, NY, USA), pp. 165–171, ACM, 1975.

[16] S. Katz and Z. Manna, "Logical Analysis of Programs," *Commun. ACM*, vol. 19, pp. 188–206, Apr. 1976.

[17] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.

[18] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints among Variables of a Program," in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Tucson, Arizona), pp. 84–97, ACM Press, New York, NY, 1978.

[19] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, 1986.

[20] J. R. Burch, E. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," in *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pp. 428–439, 1990.

[21] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear Loop Invariant Generation using Groebner Bases," 2004.

[22] E. Rodríguez-Carbonell and D. Kapur, "Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations," in *In International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04)*, pp. 266–273, ACM Press, 2004.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," in *Science of Computer Programming*, 2006.

[24] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis," in *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, (Berlin, Heidelberg), pp. 504–518, Springer-Verlag, 2007.

[25] T. Wahl, "The k-Induction Principle." 2009.

[26] A. Gupta and A. Rybalchenko, "InvGen: An Efficient Invariant Generator," in *In CAV*, 2009.

[27] A. F. Donaldson, D. Kroening, and P. Rümmer, "Automatic Analysis of DMA Races using Model Checking and k-Induction," *Form. Methods Syst. Des.*, vol. 39, pp. 83–113, Aug. 2011.

[28] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety.* New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[29] P. Cousot, "Abstract Interpretation," *ACM Comput. Surv.*, vol. 28, pp. 324–328, June 1996.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 ed., Nov. 1994.

[31] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, (New York, NY, USA), pp. 151–158, ACM, 1971.

Declaration in Lieu of an Oath

I hereby declare that I autonomously composed the present thesis and that I used none but the sources and resources stated.

This thesis has non been submitted to any other board of examiners and has not yet been published.

Passau, October 10, 2013

Matthias Dangl

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, den 10. Oktober 2013

Matthias Dangl