Scalable Abstractions for Efficient Security Checks

Doctoral Dissertation submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Aliaksei Tsitovich

under the supervision of

Prof. Natasha Sharygina

September 2011

Dissertation Committee

Dissertation accepted on 23 September 2011

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Aliaksei Tsitovich
Lugano, 23 September 2011

To my wife and parents for their limitless patience.

Abstract

Following the industrial demand to address the problem of software correctness, the computer science research community puts a lot of efforts into development of scalable and precise formal methods that are applicable to industrial-size programs. Unfortunately, most of software verification techniques suffer from the effect of combinatorial blowup also known as a "state-space explosion", i.e., situation, when the size of the system state space and, consequently, the complexity of the verification problem grows exponentially in the size of the input program. This thesis tackles this problem by development of new abstraction techniques as well as novel approaches of employing already existing ones. The results were used to construct algorithms for software verification and static analysis that discover security faults in low-level C programs.

First, this thesis presents a new algorithm that *combines precise (but slow)* abstraction method with *over-approximated (but fast)* one in the abstraction-refinement loop. It starts with the coarse over-approximated abstraction and then refines precisely, but restricts the refinement only to a subset of system state space that is related to a spurious counter-example discovered in the previous verification step of the abstraction-refinement loop. Thus, it is possible to keep the refinement computational burden low and decrease the number of required refinement iterations at the same time. We also propose a threshold-based optimization that further controls precise computation in order to avoid the unnecessary application of expensive quantifier elimination.

Second, this work defines a new technique for program abstraction. Unlike traditional program approximation approaches (e.g., abstract interpretation) it does not employ iterative fixpoint computation, instead it uses a new *summarization* algorithm that non-iteratively computes *symbolic abstract transformers* with respect to a set of abstract domains. Summaries are shorter, loop-free program fragments, which are used to substitute the original loops to obtain a conservative abstraction of the program. Our approach computes abstract transformers starting from the inner-most loop. It obtains a loop invariant by checking if the constraints defined by a chosen abstract domain are preserved by the loop. These checks are performed by means of calls to a quantifier-free decision procedure, which allows us to check (possibly infinite) sets of states with one query. Thus, unlike other approaches, our algorithm is not restricted to finite-height domains. Therefore, it allows for effective usage of problem-specific abstract domains for summarization and, as a consequence, precision of an abstract model can be tuned for specific verification needs. In particular, several memory operations-related abstract domains were applied to perform static analysis of programs for buffer overflows.

Third, this thesis addresses the problem of scalable program termination analysis. Termination of a (sequential) program can be concluded from termination of all its loops. Existing algorithms rely on iterative enumeration of all paths through a program (loop) and construction of a valid termination argument (well-founded transition invariant) for each of them using available ranking discovery methods. Instead, we present a new algorithm that applies *relational abstract domains* for *loop summarization* to discover transition invariants. Well-foundedness can be ensured either by separate decision procedure call (though it requires quantifier elimination) or by using the abstract domains that are well-founded by construction. Such a light-weight approach to termination analysis was demonstrated to be effective on a wide range of benchmarks, including the OS device drivers.

Acknowledgements

First and foremost, I wish to thank my advisor Natasha Sharygina and the colleagues I had a lucky chance to work with — Christoph M. Wintersteiger, Daniel Kroening, Stefano Tonetta, Roberto Bruttomesso and Simone Rollini. Dear friends, thank you for sharing your time, knowledge and wisdom with me.

I really appreciate the advices I got from the defense committee members: Mauro Pezzé, Orna Grumberg, Fernando Pedone and, of course, Rustan Leino whose detailed comments and discussions before and after the defense played an invaluable role in improving this work.

I am grateful to many people at University of Lugano for help, both direct and indirect. Elisa Larghi, Cristina Spinedi and Janine Caggiano with all their administrative efforts made my life at USI as smooth as it could only be. I wish to thank the doctoral students, post-doctoral researchers and professors at University of Lugano for all the good times and fruitful discussions we had together: Francesco Alberti, Domenico Bianculli, Walter Binder, Paolo Bonzini, Lásaro Camargos, Antonio Carzaniga, Giovanni Ciampaglia, Fabio Crestani, Alessandra Gorla, Cyrus Hall, Mehdi Jazayeri, Dorian Krause, Marc Marc Langheinrich, Mircea Lungu, Amirhossein Malekpour, Parvaz Mahdabi, Ilya Markov, Adina Mosincat, Vaide Narváez, Marcin Nowak, Nate Nystrom, Evanthia Papadopoulou, Marco Päsh, Edgar Pek, Nicolas Schiper, Ondřej Šerý, and Marcin Wieloch.

Last but not the least I would like to thank to all my friends in Belarus, Switzerland and all over the world, my parents for supporting me all this time, and my wife Ina for her endless patience and kindness.

Preface

This dissertation describes the results of Aliaksei Tsitovich's Ph.D. research carried out under the supervision of Prof. Natasha Sharygina at the University of Lugano (USI), Switzerland. The focus of research was on the development of techniques for software verification. Its originality lies in that it not only created new verification approaches, but also employed them in conjunction with a wide range of already-existing time-proven methods. The work resulted in the development of effective tools useful for program verification, which demonstrate the practical applicability of the new techniques. The tools are LOOPFROG [1] and OPENSMT [2, 3] and they are available for other researchers at `www.verify.inf.usi.ch/loopfrog` and `www.verify.inf.usi.ch/opensmt`.

The first contribution of this thesis is a new abstraction-refinement approach for software verification that combines precise and over-approximated techniques to achieve a synergetic effect of reduction in number of refinement iteration and, thus, the reduction of total verification time [4, 5].

The other contribution is a symbolic program abstraction technique that over-approximates loops in programs such that both safety- and liveness-related loop semantics is preserved. The approach allows for effective static analysis (i.e., safety of the memory access [6]), as well as the light-weight program termination analysis [7].

The latter work was conducted as a part of the joint project between University of Lugano and ETH Zürich funded by Swiss National Science Foundation project "Detection of Security Flaws and Vulnerabilities by Guided Model Checking" [8]. The goal of this project was the development of new model checking and static analysis techniques that aim at detection of security flaws and vulnerabilities in low-level software implementations. The project was separated into 2 interdependent parts between USI an ETHZ. The group in Lugano focused on static analysis by means of symbolic program abstraction in application to memory correctness[6, 1] and termination problems [7, 9]. The ETHZ group instead focused on ranking function computation [CKRW10] for termination analysis [10] and quantified boolean formula solving [JBS$^+$07]. The results achieved at ETHZ are presented in the thesis of Christoph M. Wintersteiger who pursued the Ph.D. degree at Computer Systems Institute of ETH Zürich.

Own publications

[1] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loopfrog: A static analyzer for ANSI-C programs. In *The 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 668–670, Auckland, New Zealand, 2009. IEEE Computer Society.

[2] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OPENSMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153, Paphos, Cyprus, 2010. Springer.

[3] Roberto Bruttomesso, Simone Fulvio Rollini, Natasha Sharygina, and Aliaksei Tsitovich. Flexible interpolation with local proof transformations. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 770 –777, San Jose, USA, 2010. IEEE Computer Society.

[4] Natasha Sharygina, Stefano Tonetta, and Aliaksei Tsitovich. The synergy of precise and fast abstractions for program verification. In *ACM symposium on Applied Computing (SAC)*, pages 566–573, New York, NY, USA, 2009. ACM.

[5] Natasha Sharygina, Stefano Tonetta, and Aliaksei Tsitovich. An abstraction refinement approach combining precise and approximated techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–14. In press / Online first.

[6] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loop summarization using abstract transformers. In *Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *Lecture Notes in Computer Science*, pages 111–125, Seoul, South Korea, 2008. Springer.

[7] Aliaksei Tsitovich, Natasha Sharygina, Christoph Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Tools*

and Algorithms for the Construction and Analysis of Systems (TACAS), volume 6605 of *Lecture Notes in Computer Science*, pages 81–95. Springer, Saarbrücken, Germany, 2011.

[8] Aliaksei Tsitovich. Detection of security vulnerabilities using guided model checking. In *International Conference on Logic Programming (ICLP)*, volume 5366 of *Lecture Notes in Computer Science*, pages 822–823, Udine, Italy, 2008. Springer. Extended abstract for doctoral consortium.

[9] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loopfrog — loop summarization for static analysis. In *Proceeding of the Workshop on Invariant Generation (WING)*. Easychair, 2010. In press.

[10] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *International Conference on Computer-Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 89–103, Edinburgh, UK, 2010. Springer.

Contents

Chapter 1

Introduction

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C. A. R. Hoare

1.1 The necessity of software verification

The modern information-driven society demonstrates an obvious need to have adequate security measures that can safeguard the storage and transfer of sensitive information. Software performs almost 100% of manipulations on information around us and, thus, should be placed under particularly strict security control. Unfortunately, writing a program without bugs is still one of the greatest challenges for developers. As a result, low-level implementation mistakes lead not only to malfunctioning software but also to security breaches.

Verification of software using formal methods is recognized as a candidate to address the problem. Mathematical reasoning gives a guarantee of 100% confidence in correctness of analysis results. However, the formal methods-based approach faces multiple challenges when it comes to application in industry. Currently substantial research efforts are dedicated to both theoretical and practical aspects of formal verification. The grand challenge for research in formal methods and formal verification tools is to ensure software reliability, trustworthiness and robustness [Hoa03, Cou01].

1.2 Major approaches to software verification

Verification by model checking One of the most popular and well-studied verification instruments today is *model checking* — the technique that attempts an exhaustive search of the entire state space of a system for violations of a property of interest. Introduced in 1981 [CE81, QS82], it is one of the most commonly used formal verification techniques in a commercial setting. In 2007, a Turing Award was given to the inventors of model checking (Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis) as a commemoration of its importance.

To perform model checking one starts with a construction of a finite state machine for a given system, where nodes are possible variable states and edges are transitions among them. System executions, for which a property of interest is violated, can be identified by particular variable values (or a sequence of variables valuations). All possible executions of the finite state machine are analyzed using some form of graph exploration algorithms. A big advantage of model checking is that it produces a *counterexample* if the violation of the property is found, i.e., there is step-wise evidence of system execution that leads to a violation state. Such a counterexample is extremely helpful for diagnosis of the error.

Verification of hardware designs, inherently finite in its nature, is the most straightforward application of model checking. Software systems are more complex since they can potentially have an infinite number of executions. Nevertheless, techniques that enable software model checking are also available, although they are not complete — the model checking tool is not guaranteed to always terminate with a positive or negative answer. Practical applications of software model checking are also heavily limited by a combinatorial blow up of the system state-space representation, a phenomenon known as *state-space explosion*. That is, the size of the system state space grows exponentially in number and size of program variables. The task of explicit state-space exploration quickly exceeds the capacity of the tool. Therefore, the main research challenge of practical software model checking is to address the scalability problem.

Throughout the history of model checking research, a number of methods were proposed for scalability improvement. *Symbolic model checking* [McM93] was a breakthrough when Kenneth McMillan proposed using Binary Decision

Diagrams for the symbolic representation of a system state-space, instead of the previously common explicit state-space representation. For software model checking, it raised the applicability level from hundreds to thousands lines of code. Also, substantial improvement was proposed by Doron Peled[PP90] and Patrice Godefroid [God90] who noticed that the size of the system state-space can be reduced based on partial order relation between the transitions in a system: indeed, it is common for asynchronous systems (which are popular models of software), that certain transitions in a system cannot be taken before the others.

Bounded model checking Further improvement of verification scalability, from thousands to tenth of thousands lines of verified code, was achieved with the help of another prominent technique — *Bounded Model Checking* (BMC) proposed by Biere and others [BCCZ99]. BMC does not aim to verify all possible program executions, but only those of limited length. Starting from the initial state algorithm unrolls the program for a fixed number of steps and checks whether a property is violated in all the executions of this fixed length. This process terminates either if the underlying decision procedure exceeds its time or memory bounds, or if a counterexample is found.

Notably, the presence of loops is the major obstacle for the successful application of BMC to program verification. Each loop needs to be explicitly unrolled, or *unwound*, so that all possible paths through the loop are considered. If the loop termination or loop bound cannot be established, BMC cannot progress beyond the loop, forced to unroll (infinitely) many loop iterations.

Abstraction The research results presented in this thesis are related to the most promising techniques used to combat the "state-space explosion" of program verification : *abstraction* and *abstraction refinement*-based approaches.

The term of *abstraction* [CGL94] groups the family of techniques that reduce the state-space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves all behaviors of the original system. The latter property of the abstract system is called *soundness*, i.e., no original behaviors are lost. However, abstraction can introduce new behaviors that are impossible in the original system.

The value of abstraction for software verification is twofold. First of all, it allows the possibility to obtain a finite representation of a program, while the state-space of the original program can be infinite. Second, it reduces the size of the model for analysis. The right abstraction choice is a key to a scalable verification.

One way to incorporate abstraction in a verification algorithm is to combine it with *refinement*. Refinement is applied to an abstract model in order to remove any unrealistic, spurious, behavior that was found during verification (model-checking) of an abstract system. Since refinement can be applied several times, this approach is also known as an *abstraction-refinement loop* [Kur95]. When refinement is based on the counterexample obtained from model-checking of an abstract system, the technique is also named *Counterexample-Guided Abstraction-Refinement* (CEGAR).

Abstraction exploits concepts of *abstract interpretation*, first proposed by Patrick and Radhia Cousot [CC77]. It is an approach for construction and evaluation of abstract systems, which builds sound (over-)approximations of computer programs. A program is (partially) executed with abstract variable values instead of concrete ones, such that program semantics is preserved with respect to an *abstract domain*. The process is iteratively repeated until a fixpoint is reached. To guarantee convergence of the fixpoint computation in programs with loops or recursive procedures an extrapolation technique is used, called *widening* — an over-approximation of a set of abstract values.

Examples of abstract domains include interval arithmetic [CC77], which abstracts variables' values to intervals, and the octagon domain [Min06], which uses a constraint representation in the form $\pm v_1 \pm v_2 \leq c$, that, in two dimensions, coincides with the set of eight-sided polyhedra. Other examples are octahedra [CC04], convex polyhedra [CH78a, HPR97], difference-bound matrices [Min01] and linear inequalities [SKH03].

The models constructed by abstract interpretation can be used for program analysis either by itself or in combination with techniques like model checking [CC99]. A number of static analysis tools are based on abstract interpretation (e.g., ASTRÉE [CCF+05]). Such analysis is sound, i.e., it does not report that a property holds when it does not hold in the concrete program. However, abstraction often leads to false positives — it is reported that a property is violated, while in fact it does hold in the concrete program. In particular the aggressive widening required to achieve scalable fixpoint computation leads to an increase in the number of false positives. The possibility of false positives makes the technique *incomplete*.

1.3 Challenges in software verification

Finding the right abstraction is a key to further extension of the applicability of formal methods to real problems of software and hardware engineering. Building a concise (but yet manageable) model that precisely represents the semantics of a system is an objective of research in this area, including work of this thesis.

Practical application of numerous theoretical abstraction frameworks often faces the wall of inefficiency when it comes to real systems. The size of the formal model that is constructed for the pair program/property, and the complexity of the following analysis depends on a number of factors. First of all, the logic used to encode the problem and the algorithm that validates the problem in this logic defines the theoretical complexity. Secondly, the number of variables in the program and their data types set the boundaries of the state space. Next, the control flow of the program in combination with program variables defines the possible behaviors (executions) of the program. Both number of possible executions and the state space can be often infinite in real systems. The ability to manage all these dimensions of complexity defines the *scalability* of each technique.

It is worth noting that, in contrast to software engineering, where scalability often means being able to treat proportionally more lines of code, in verification the scalability term refers to being able to reason (soundly) about bigger state spaces and larger sets of behaviors. Although there is a connection between size of the program and its state-space, the correspondence between these two notions is not straightforward. Even a program of 10 lines can exhibit infinite number of executions and can have a state space of millions of states. In particular, the presence of multiple nested loops in a program makes it complex to verify all its possible behaviors. Figure 1.1 demonstrates the complexity issues on several program examples and suggests a simple abstraction that enables scalable buffer access verification in all of the programs.

The specific challenges addressed in this thesis were motivated by a simple practical observation: formal tools often “loop around” a particular location in the program, trying to “clarify” its formal meaning in order to conclude if it can lead to a bug or not. This behavior varies in representation for different techniques: bounded model checking cannot unwind infinite loops without losing soundness; abstract interpretation over-approximates such locations with aggressive widening that causes imprecise abstract values; abstraction refinement struggles to find the right predicates; invariant discovery methods are not applicable to complex data manipulation, etc. This thesis tackles the challenges

```
#define SIZE 8
int c[SIZE];
int main() {
 int i;
 for(i=0;i<SIZE;++i)
  c[i]=i+1;
}
```

$2^4 + 1$ states, 1 execution

```
#define SIZE 256
int c[SIZE];
int main() {
 int i;
 for(i=0;i<SIZE;++i)
  c[i]=i+1;
}
```

$2^9 + 1$ states, 1 execution

```
#define SIZE 256
int c[SIZE][SIZE];
int main(int argc, char  argv[])
{
 int i,j;
 for(i=0;i<SIZE;++i)
  for(j=0;j<SIZE;++j)
   c[i][j]=argc>1 ? argv[1][0] : 0;
}
```

$2^{10} + 1$ states, 2^8 executions

```
#define SIZE 256
int c[SIZE];
int main() {
 int i=0;
 do {
  if (i>=SIZE) i=0;
  c[i]=getchar();
 } while (c[i++] != ’.’);
}
```

2^{24} states, infinite executions

Figure 1.1. Verification of access correctness to the buffer c in all these programs can be done, for instance, by analysis of the abstract system constructed using only two predicates: $i \geq 0$ and $i < SIZE$

of the practical application of formal methods by proposing new algorithms for efficient verification. In particular, this thesis addresses the problems of 1) reducing the number of abstraction-refinement iterations needed for program verification and 2) scalable construction of loop over-approximations that allow preservation of both safety- and liveness-related semantics of a loop.

Problems with abstraction refinement-based verification

A low number of abstraction refinement iterations is fundamental for the success of the CEGAR loop and, consequently, to overall verification effort, especially when applied to industrial benchmarks. In fact, the number of predicates required to verify the property grows with the complexity of a system. Furthermore, the time spent in model checking steps grows exponentially with the number of predicates. For this reason, it is of paramount importance to avoid as many redundant iterations as possible: even a single saved iteration can result in a substantial time-saving for large systems.

Verification of programs with loops

Loops in programs are the Achilles' heel of program verification. Sound analysis of all program paths through loops requires either an explicit unwinding or an over-approximation (of an invariant) of the loop.

Unwinding is computationally too expensive for many industrial programs. For instance, loops greatly limit the applicability of bounded model checking. In practice, if the bound on the number of loop iterations cannot be precomputed (the problem is undecidable by itself), BMC tools simply unwind the loop a finite number of times, thus trading the soundness of the analysis for scalability.

At the same time computation of sufficiently strong invariants for loops is an art. Abstract interpretation and CEGAR rely on saturating procedures to compute over-approximations of the loop. For complex programs, this procedure may require many iterations until the fixpoint is reached or the right set of predicates is determined. Widening is a remedy for this problem, but it introduces further imprecision, yielding spurious behavior.

Thus, it can be concluded that, loop over-approximation techniques that are predictably fast, scalable and precise are essential for program verification.

Verification in the presence of (possibly) infinite loops

Furthermore, construction of program over-approximation often assumes that every path through a loop terminates, i.e., eventually it leaves the loop after a finite number of iterations. Unfortunately, it is not the case for many real applications (some loops are even designed to be infinite). Thus, careful reasoning about loop (non-) termination should be addressed by abstraction techniques as well.

1.4 Thesis statement and contributions

This dissertation, establishes the following theses:

1. Computationally expensive precise program abstraction can be effectively incorporated in the abstraction-refinement loop if synergetically interwoven with its fast, approximated counterpart. The precise computation pays off if applied not to a whole set of predicates, but only to those related to a discovered spurious counter-example. This way, it is possible to keep the refinement computational burden low and to decrease the number of required refinement iterations at the same time. In addition, the on(off)-line computed threshold of expensive precise computation can help to leverage its cost in the overall verification process.

2. Programs with loops can be abstracted with a controlled level of precision without iterative fixpoint computation or explicit loop unwinding. Abstract transformers can be constructed to over-approximate program fragments with infinitely many behaviors in a way that requires a finite number of satisfiability checks. The resulting models can encode both safety- and liveness-related semantics of the original program and, thus, can be used to verify properties of both classes, e.g., buffer access correctness, program termination, etc.

3. Termination of a (sequential) program can be concluded from termination of all its loops. Program abstraction via loop summarization is able to preserve termination property of over-approximated loops. For that, one should employ an abstract domain capable of encoding the relation the between pre- and post states of a loop iteration, thus, enabling the discovery of disjunctively well-founded transition invariants for a loop.

These theses are addressed as follows:

1.4.1 Reducing CEGAR iterations by combining fast and precise abstraction

Contribution We distinguish between two classes of abstraction methods — *precise* and *fast*. The first enables only those transitions between abstract states that correspond to transitions in the concrete system. The second allows additional transitions, i.e., it over-approximates the set of transitions, causing more spurious behaviors in the abstract system. The first is usually expensive in computation because it requires explicit quantification of the transitions set. The second can be effectively implemented without quantification, but requires more refinement steps to remove all spurious behaviors.

This thesis finds a balance between the two types of abstraction by applying a fast abstraction first and then refining precisely only the subset of a system that is related to the detected counterexample. This way, the cost of the precise computation in the refinement is kept relatively low, but we achieve an important reduction in the number of refinement iterations that lead to a decrease of the total verification time.

Practical application We implemented the abstraction synergy algorithm using the SatAbs CEGAR-based model-checker for ANSI-C programs [CKSY04] and demonstrated with experiments the practical value of the proposed approach. The combined method uniformly outperforms both precise and fast counterparts on a wide set of benchmarks.

The technique is described in detail in Chapter 3 and was presented at the 24th Annual ACM Symposium on Applied Computing (SAC 2009) [4]. It was also further extended to be published in the International Journal on Software Tools for Technology Transfer (STTT) [5]. Experimental results and the tool are available at: `www.verify.inf.usi.ch/projects/synergy`.

1.4.2 Symbolic program summarization using abstract transformers

Contribution We focus on efficiency of program abstraction and, to that end, we propose an algorithm that replaces program fragments by their symbolic abstract transformers. Specifically, for programs with no loops, an algorithm precisely encodes the program semantics into symbolic formulæ. For loops, the abstract transformers are constructed based on the problem-specific abstract domains. The approach does not rely on fixpoint computation of the abstract transformer and, instead, builds the latter as follows: an abstract domain is used to draw a candidate abstract transition relation, which is checked to be consistent with the semantics of the loop.

The algorithm allows tailoring the abstraction to each program fragment. Also, it avoids the possibly expensive fixpoint computation of the abstract transformer and relies on the finite number of relatively simple consistency checks. Those are performed by means of calls to a SAT(SMT)-based decision procedure, which enables the checking of (possibly infinite) sets of states within one query. Thus, the algorithm is not restricted to finite-height domains.

Practical application We implemented our loop summarization technique in a tool called LOOPFROG. We validated the new approach on a large set of ANSI-C benchmarks using abstract domains tailored to the discovery of buffer overflows. Our experience demonstrates the applicability of the approach to industrial code and its advantage over fixpoint-based static analysis tools.

The algorithm is described in Chapter 4 and also appeared in the paper presented at the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008) [6]. LOOPFROG was demonstrated at the 24th IEEE/ACM International Conference on Automated Software Engineering [1] and now is available at: `www.verify.inf.usi.ch/loopfrog`.

1.4.3 Program termination analysis using loop summarization

Contribution Our loop summarization technique (briefed in Section 1.4.2) is a general-purpose loop and function summarization method. This dissertation further employed it to address the problem of program termination. We extended it with relational abstract domains to discover (disjunctively well-founded) *transition invariants* — valid relations between pre- and poststates of a loop. If a disjunctively well-founded transition invariant exists for a loop, we can conclude that it is terminating, i.e., any execution through a loop contains a finite number of loop iterations. We also exploited *compositionality* of transition invariants to limit analysis to a single loop iteration.

Practical application We implemented loop termination analysis in the LOOPFROG static analyzer. Due to the fact that the safety checker is employed to analyze only a single unwinding of a loop at any point, we gain large speedups compared state-of-the-art tools that are based on path enumeration. At the same time, the false-positive rate of our algorithm is very low in practice, which we demonstrate in experimental evaluation on a large set of Windows device drivers.

The algorithm and results are detailed in Chapter 5; the work was also presented at the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011) [7]. The implementation is available within the functionality of the LOOPFROG tool: `http://www.verify.inf.usi.ch/loopfrog`.

1.5 Outline

The dissertation is structured as follows. Chapter 2 introduces the required concepts and notation. Chapter 3 presents a technique to combine fast and precise abstractions. Chapter 4 describes program abstraction via loop summarization and discusses the practical applicability of this approach to program analysis. The algorithm is extended in Chapter 5 to work with transition invariants that allow a termination analysis technique to be built on top of it. Related work is discussed at the end of each chapter. Finally, Chapter 6 states the conclusion of the work.

Chapter 2

Background, Concepts and Notation

The noblest pleasure is the joy of understanding.

Leonardo da Vinci

This chapter introduces the notation required to present the results of the dissertation work, and provides an overview of the state of the art in the relevant fields of program verification.

2.1 Program modelling

Transition systems

To reason formally about a software or hardware system one should first build a formal model of it. *Transition systems* (TS) are the most common models used for the faithful representation of system behavior. Intuitive understanding of a transition system is quite simple: it is a directed graph with nodes that represent program *states* and edges that reflect *transitions* among states.

Definition 1. *A transition system is a tuple $\langle S, I, R \rangle$, where:*

- *S is a set of states;*
- *$I \subseteq S$ is the set of initial states;*
- *$R \subseteq S \times S$ is the transition relation.*

A *path* of a transition system is a (possibly infinite) sequence of states s_0, $s_1, \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

A *prefix* is a finite path that starts in an initial state, i.e., $s_0 \in I$.

A state $s_i \in S$ is *reachable* if there exists a prefix that ends with it.

A path is *maximal* if it ends in a terminal state (state with no outgoing transitions) or is infinite.

An *execution* of a transition system is a maximal path that starts in an initial state.

We use a relational composition operator $\circ$ which is defined for two binary relations $R_i \subseteq S \times S$ and $R_j \subseteq S \times S$ as

$$R_i \circ R_j := \left\{ (s, s') \in S \times S \;\middle|\; \exists s'' \in S \,.\, (s, s'') \in R_i \wedge (s'', s') \in R_j \right\} .$$

To simplify the presentation, we also define $R^1 := R$ and $R^n := R^{n-1} \circ R$ for any relation $R : S \times S$.

Note that a relation R is transitive if it is closed under relational composition, i.e., when $R \circ R \subseteq R$. The reflexive and non-reflexive transitive closures of R are denoted as R^* and R^+ respectively. The set of reachable states is then defined as $R^*(I) := \{s \in S \,|\, \exists s' \in I \,.\, (s', s) \in R^*\}$.

Although the TS of Definition 1 is powerful enough to model the behavior of any hardware or software system, it is distant from the design primitives of modern systems: there are no explicit notions of variables, commands or program structure. Therefore, to facilitate reasoning about programs, the formalism describing transition systems can be enriched with other elements, such as:

- set of actions Act — to describe commands that relate the states (transition relation is then defined as $R \subseteq S \times Act \times S$);
- set of atomic propositions AP — to represent some facts about a program, e.g., variable values;
- labeling function $L : S \rightarrow 2^{AP}$ — to map variable values to states of a program;

A transition system is considered *finite* if S, Act and AP are finite.

An alternative way to include the notion of program variables into a transition system is to use a set of state variables V. Then the set of states S is implicitly given by all assignments to the variables of V. The transition relation is then defined as a formula over the variables of V. Let us use V' in pair with V to denote the set of next state variables, i.e., $v' \in V'$ represents the next value of $v \in V$. The set of transitions is then denoted as $T(V, V')$. We use $in(t)$ and $out(t)$ to denote respectively pre-state and post-state of a transition $t \in T(V, V')$. Given a formula ϕ, we write $\phi[V/V']$ to denote the result of substituting every free occurrence of every variable $v' \in V'$ with its corresponding v. We use $\exists V(\phi)$ to denote the existential quantification of every variable in V. We also use $\exists V \setminus \{v\}(T)$ to denote the existential quantification of every variable in V except from v (whose occurrences remain free).

Altogether it gives us another way to define the transition system:

Definition 2. *A transition system is a tuple $M = \langle V, I, T \rangle$, where*

- *V is a set of variables;*
- *$I(V)$ is a formula that represents the initial states;*
- *$T(V, V')$ is a formula that represents the transitions.*

A state s is initial iff $s \models I(V)$. Given two states s_1 and s_2, there exists a transition t between s_1 and s_2 iff $s_1, s_2' \models T(V, V')$.

A path of M is a finite sequence π of transitions $t_0, t_1, ..., t_n$ such that $in(t_0) \models I$, and, for every $0 \leq i < n$, $out(t_i) = in(t_{i+1})$. In general, given a transition relation formula T and a path π, $\pi \models T$ is used to denote that $\pi[i] \models T$ for every $0 \leq i \leq |\pi|$.

Program graph

Another program modeling structure — *program graph* — allows explicit reasoning about the program states to be avoided. Instead, it uses program locations as nodes and program commands as edges that connect locations. Formally:

Definition 3. *A program graph is a tuple $G = \langle PL, E, pl_i, pl_o, L, C \rangle$, where*

- *PL is a finite non-empty set of vertices called* program locations*;*
- *$pl_i \in PL$ is the initial location;*
- *$pl_o \in PL$ is the final location;*

```
p=a;
while(*p!=0){
 if(*p=='/')
   *p=0;
 p++;
}
```

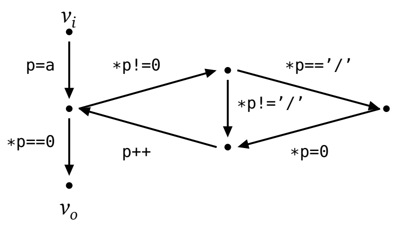


Figure 2.1. The example of a program and its program graph

- *$E \subseteq PL \times PL$ is a non-empty set of edges; E^* denotes the set of* paths*, i.e., the set of finite sequences of edges;*
- *L is a set of elementary commands;*
- *$C : E \to L$ associates a command with each edge.*

A program graph is often used as an intermediate modeling structure in program analysis. In particular, it is used to represent a control-flow graph of a program. It reflects all possible paths in a program but does not represent all concrete variable valuations and, thus, just by itself, it is not enough to model a program. It needs to be paired with concrete variables to depict the program semantics.

Example 2.1. *To demonstrate the notion of a program graph we use the program fragment in Figure 2.1 as an example. On the left-hand side, we provide the program written in programming language C. On the right-hand side, we depict its program graph.*

Let $dom(v)$ for $v \in V$ denote the set of possible values of a variable v, the domain of v.

Then let U denote the universe where the values of the program variables are drawn, a union of the domains of all variables in V, i.e.,

$$U := \bigcup_{v \in V} dom(v)$$

The set of commands L consists of tests L_T and assignments L_A, i.e., $L = L_T \dot{\cup} L_A$, where:

- a test $q \in L_T$ is a predicate over $dom(q) \subseteq U$;
- an assignment $e \in L_A$ is a total map from $dom(e) \subseteq U$ to U.

A program P is then formalized as the pair $\langle U, G \rangle$, where U is the universe and G is a program graph. We write L^* for the set of sequences of commands. Given a program P, the set $paths(P) \subseteq L^*$ contains the sequence $C(e_1), \ldots, C(e_n)$ for every $\langle e_1, .., e_n \rangle \in E^*$.

State-space explosion

We can relate modeling using program graphs and the transition system: in order to obtain a $TS(G)$ we apply commands L^* to variables values of U, i.e., we *unfold* a program graph with variables to a transition system:

- $S = PL \times dom(v)$;
- $I = \{pl_i\} \times dom(v)$;
- $R = S \times E \times S$;

Even if we assume the program graph to be finite[1], the unfolded transition system often can be infinite (in case of dynamic memory allocation) or at least significantly bigger (in number of nodes and edges) than the program graph. For instance, if we limit the domain of the variable a in the program graph of Example 2.1 to all strings of 5 characters, the image of the transition system will hardly fit on the page. Provided a pointer p can point to 2^{32} different locations, a buffer a can have $10 * 2^8$ different elements and we have 6 program locations we end up with a set of states S of size $6 \times (2^{32}) \times 5 \times (2^8) = 3.29853488 \times 10^{13}$.

The growth of a transition system corresponding to example 2.1 is an instance of a combinatorial blow up, *state-space explosion*, which happens when explicit transition system is constructed from a program graph. Clearly, explicit representation of a set of states is impossible for any realistic program analysis scenario.

Although, not every state of S is reachable from I following the transitions of R. In the unfolded transition system only those states are connected that correspond to program locations originally joined by edges E of a program graph. Notably, loops in programs have a big impact on the resulting size of reachable subset of S as they cause mapping of a single program location to $|dom(v)|$ reachable states. Therefore loops are identified as the major obstacles in practical program analysis.

[1]This assumption holds for many existing languages, though there are several (mostly scripting) ones that allow dynamic modification of the program text and, thus, corresponding program graph can be infinite

The later chapters of this dissertation make use of both transition systems and program graphs to present algorithms that were designed to cope with the state-space explosion.

2.2 Properties specification

After the system behavior is formalized and prior to its analysis, we need to define an important element of the verification process — a property of interest. The specified property answers a question: what does it mean for the system to be correct? System correctness (or equivalently incorrectness) is then defined with regard to properties, which hold (or may not) for the analyzed system.

We briefly introduce the most important classes of properties, namely safety and liveness, and elaborate on the particular subclasses that are the main focus for the dissertation:

Safety

The properties, grouped in the safety category, can be characterized informally as "nothing bad should happen" [Lam77], i.e., they aim to guarantee the absence of some bad behavior in the system. Formally it is defined as follows:

Definition 4. *A property ψ over V^* is called a safety property iff for each path σ where ψ does not hold, there exists a finite* bad prefix *$\hat{\sigma}$ that can not be extended to hold in ψ.*

A classical example of a safety property in the context of security vulnerabilities is correctness of memory access. It can be formulated as "a program never accesses memory beyond allocated space".

Liveness

The informal definition of liveness sounds like: "something good should eventually happen" [Lam77], i.e., if a liveness property holds, the system is guaranteed to reach some state of interest. Formally:

Definition 5. *A property ψ over V^* is called a liveness property iff every finite prefix $\hat{\sigma}$ can be extended in to an execution were ψ holds.*

Program *termination* is a classical example of a liveness property: a program is terminating iff it does not have infinite executions.

Note, that separation into safety and liveness classes is not exclusive. The correctness of the system maybe defined by the property which is of both classes at the same time. For example: "a correct program always terminates and does not access non-allocated memory".

2.2.1 Invariants

Many interesting properties are grouped under the notion of *invariants*. Informally, an invariant is a property that always holds for (a part of) the analyzed system. The notion of invariants in computer science has been actively used since it was introduced by C. A. R. Hoare in his work about the logic rules for reasoning about program correctness [Hoa69]. A naïve algorithm to check if some formula ψ is an invariant of a system M boils down to a complete depth-first or breadth-first graph exploration with an invariant check performed for every visited state or transition.

Important types of invariants that are distinguished and used in this work are *state invariants*, *transition invariants* and *loop invariants*.

State invariants

Definition 6. *Formula ψ over V of a transition system $M = \langle V, I, T \rangle$ is a state invariant for M iff ψ holds in all reachable states of M (including all initial states).*

A naïve procedure that checks, if some ψ is a state invariant of a system M, terminates iff the M is finite, i.e., we can explore all states of M and check if ψ holds for them. Thus, state invariant checking corresponds to verification of a safety property.

In terms of relations, a state invariant is defined as a superset of a set of reachable states, i.e., $R^*(I) \subseteq \hat{S}$.

The state invariant was the first important type of invariant put in service for program analysis. For historical reasons, it is often referred to simply as an invariant; we, however, will explicitly call it a state invariant when it is required to stress its difference with other classes of invariants.

Transition Invariants

In contrast to state invariants that represent the safety class of properties, transition invariants, introduced by Podelski and Rybalchenko [PR04b], enable reasoning about liveness properties and, in particular, about program termination.

A program is terminating if it does not allow infinite computations, which follows from the well-foundedness of the transition relation (restricted to the reachable states). A *well-founded* relation is a relation that does not contain infinite descending chains or, in other words:

Definition 7 (Well-foundedness). *A relation R is* well-founded (wf.) *over S if for any non-empty subset of S, there exists a minimal element (with respect to R), i.e., $\forall X \subseteq S \,.\, X \neq \emptyset \implies \exists m \in X, \forall s \in S (s, m) \notin R$.*

The same does not hold true for the weaker notion of *disjunctive well-foundedness*. However, Podelski and Rybalchenko show that disjunctive well-foundedness of a *transition invariant* is equivalent to program termination:

Definition 8 (Disjunctive Well-foundedness [PR04b]). *A relation T is* disjunctively well-founded (d.wf.) *if it is a finite union $T = T_1 \cup \cdots \cup T_n$ of well-founded relations.*

Definition 9 (Transition Invariant [PR04b]). *A* transition invariant *T for program P represented by a transition system $\langle S, I, R\rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.*

Termination is can then be defined as:

Theorem 1 (Termination [PR04b]). *A program P is terminating iff there exists a d.wf. transition invariant for P.*

Discovery of d.wf. transition invariants is discussed later in Chapter 5, including the technique for program termination analysis reported in this dissertation.

Loop invariants

To simplify reasoning about loops — the major obstacles in practical program analysis — a special kind of invariant is used — *loop invariant* [Hoa69]. A loop invariant is a formula ψ that holds on entry into a loop and on every iteration of the loop. As a result, it is guaranteed to hold immediately on exit of the loop[2]. For example, the loop in Figure 2.1 can have a loop invariant "`p` $\leq$ `length(a)`".

Discovering loop invariants proved to be a non-trivial problem through the 40 year history of active research on the subject [CH78a, HH95, BL99,

[2]Here we consider structured loops or, at least, loops, for which the only way to break out of it is before an iteration has changed the state of variables.

SSM08]. The early work in this area was mostly about heuristically-based methods [KM73, Weg73]. The methods were incomplete and practically inapplicable because of the lack of computational power. A complete approach was developed later based upon the mechanical derivation of a loop invariants technique [CP93], which derived logical consequences of first-order formulæ. These techniques, however, turns out to be computationally demanding and also require inductive lemmas to be provided manually.

The increased availability of computing resources and advances in symbolic analysis brought attention back to automated invariant generation. A number of iterative methods were successfully applied for invariant discovery, including abstract interpretation [CH78a, BL99], abstraction refinement methods and constraint colving [SSM04, RCK07].

Note, that both state and transition invariants can be used as loop invariants, once we consider a loop of a program as a separate sub-program. Both state and transition invariants are employed in this dissertation to construct sound over-approximations of program fragments. This algorithm is detailed in Chapter 4.

2.2.2 Security flaws and vulnerabilities

There is no exact unified definition of what a security vulnerability is. The understanding of what it means for software to be secure varies between research fields and industrial settings. The focus of this work is on the analysis of real software, in particular, of programs written in ANSI-C. Therefore, we target possible low-level errors in code — both accidentally or maliciously planted. It should be noted that, though security analysts are mainly interested in exploitation of errors, we focus on technical details (i.e., buffer overflows) that allow formal encoding of an error as a property for verification.

Buffer overflows

The SANS Software Security Institute [SAN] lists *buffer overflows* as one of the three major programming errors which account for more than 85% of the critical security vulnerabilities in last decade. Nearly half of the SANS Top-25 attack targets of the 2009 are due to buffer overflow vulnerabilities. Buffer overflow vulnerabilities remain a constantly increasing source of security vulnerabilities in software systems, and thus an unsolved problem.

SANS also reports that the attacker community does not mainly target operating systems anymore, but also applications in popular areas such as media players, anti-virus systems, back-up systems, and office applications.

Furthermore, IBM's ISS X-Force Team reports [IBM07] that 88.4% of the 7,247 vulnerabilities discovered in 2006 (a 40% increase over 2005) could be exploited remotely. Half of all vulnerabilities (50.6%) would allow an attacker to gain access to the host after successful exploitation. A further 11% allow for denial of service attacks. These numbers further increased in their 2007 Midyear Report: 51.6% allow gaining access and 13.4% allow denial of service attacks, and 90% of all vulnerabilities can be exploited remotely. To a large degree, these vulnerabilities can be blamed on buffer overflows.

Buffer overflows occur whenever a program loop runs over some part of memory, failing to stop before the end of the memory block. In languages like C and C++, this is usually due to zero-terminated string buffers. Overrunning loops that write to strings, or data arrays in general, are the most dangerous threat: Often they allow an attacker to execute arbitrary, injected code. Buffer overflows came to fame when "Aleph One" published a tutorial on how to exploit them in "Phrack Magazine" [Ale96], an online Hacker journal. Further details of buffer overflow attacks can, for example, be found in the SANS online reading room [Don02].

A loop that reads a string from memory and returns this data to the user might reveal confidential information to an attacker when the string memory is overrun. More often, the system might simply crash because of segmentation faults — the desired outcome of a denial of service attack.

This is possible, if the attacker can control data that is being written in the over-running loop. By supplying excess data, variables that reside in memory locations after the string (or before, for buffer underflows) can be overwritten with attacker controlled data. The most common exploitation technique is to overwrite a function's return value on the stack with a pointer into the injected data. In contrast to a stack-based exploitation technique, many heap-based techniques are known as well. For example, attackers can overwrite function pointers that are held in dynamically allocated memory on the heap.

Buffer overflow is one of the most important bugs that should be addressed by software analysis techniques. For this reason we focus on practical evaluation of the algorithm via analysis of this type of vulnerabilities as described in Chapter 4. However, the presented techniques are general and can be applied to verify other security-related properties as well, including numeric under and over-flows, input sanitization, etc.

2.3 Model checking

Once a system model and a property are defined, the verification itself can be performed. If someone is looking to perform a verification task automatically, the most probable candidate will be *model checking*. Techniques gathered under the guise of this name allow exhaustive search of the entire state space of a system for violations of a property of interest.

Given a transition system M and a property ψ, a model checking problem is to determine whether all system executions satisfy the property, i.e., $M \models_\forall \psi$ (universal quantifier $\forall$ denotes that all executions should be satisfied). The effective solution usually takes into account the characteristics of a pair M and ψ such as symbolic or explicit representation of M, logic of ψ, etc. The type of the property also plays an important role as different algorithms are used to verify different classes of properties (i.e., safety and liveness, or state and transition invariant).

An important symbolic model checking algorithm is *Bounded Model Checking* (BMC) [BCCZ99]. BMC does not aim to verify all possible program execution, but only those of length k or less. Starting from the initial state, the algorithm unrolls the program for a fixed number of steps and checks whether a property is violated in all executions of this fixed length. It is achieved by:

- constructing a symbolic representation formula that encodes possible state transformations along the transitions of all k-paths;
- constructing a formula that reflects the projection of a property ϕ to unwound transitions;
- performing a satisfiability check if there exists an execution that does not fulfill the property, i.e., $M \not\models_k \phi$.

In the case when ϕ is a safety property, the formula is satisfiable iff there exists a counterexample of length k. If not so, k is increased to search for longer counterexamples. This process terminates either if the underlying decision procedure exceeds its time or memory bounds, a counterexample is found, or k exceeds a *completeness threshold* [KS03]. In the latter case, k is sufficiently large to ensure that no counterexample exists, and thus, it can be concluded that $M \models_\forall \phi$.

Depending on the formula encoding, different decision procedures can be used for satisfiability checks in BMC. When BMC was originally proposed it relied on a Boolean encoding and a propositional solving procedure. That is, a *Boolean satisfiability problem* [DPG97] (also known as SAT-problem) is constructed — the problem of determining if the variables of a given Boolean formula can be assigned in a way that makes the formula evaluate to $TRUE$. Special tools, SAT-Solvers, are used to find an answer in this case. SAT-Solvers underwent a period of rapid growth starting from the late 90th. Effective implementations like Grasp [SS96], Chaff [MMZ$^+$01] and MiniSAT [ES03] raised the level of machine-solvable formulæ to millions of Boolean variables. As a consequence, BMC, as well as the other verification techniques that rely on SAT, benefited from this progress.

The recent advances in research on decision procedures allow replacing propositional solvers with new computational engines called SMT-solvers (Satisfiability Modulo Theories). In SMT, the formulæ used for path representation are no longer limited to Boolean variables but can also include first-order constraints in decidable theories of equalities, bit-vectors, arrays, linear arithmetic, etc. First, this enables more compact and simple symbolic encoding of program models. Second, specialized decision procedures for each theory and their combination speed up the satisfiability checks. Examples of SMT-Solvers in active development are Yices [Yic], Z3 [Z3], Mathsat [MSA], OpenSMT [BPST10].

2.4 Automated theorem proving

SMT-solvers are often recognized as representatives of another major class of tools that employ math proofs to reason about formal specifications — Automated Theorem Provers (ATP). ATP is designed to show if some statement (the conjecture or the property) is a logical consequence of a set of statements (the axioms and hypotheses or the specification). ATP attempts to construct the proof automatically, although it is not always possible, i.e., available axioms might be not enough to conclude on validity of the conjecture. In this case user input in form of additional lemmas can be of use. Interactive Theorem Provers are a specialized class of ATP designed for effective interaction with the user.

ATP systems are used in a wide variety of domains including applications to program verification. Given an appropriate formulation of the problem as axioms, hypotheses, and a conjecture, it can be employed to performs steps of verification process (e.g., abstraction or counter-example simulation) or to do the verification job completely. The latter is, however, less frequent due to computational inefficiency of the generalized methods for math proof constructions compared to techniques specifically designed for program verification. There various model checking techniques and ATP are often combined to achieve better performance.

Examples of ATP tools in active development are ACL2 [Kau98], Coq [BC04], HOL [NPW02], Vampire [RV99]. Industrial use of automated theorem proving is mostly concentrated in integrated circuit design and verification. A discovery of a division bug in Pentium P5 forced main players on the market to invest into techniques that can minimize the risk of repetition of a similar case. Since then, complicated arithmetical units of modern microprocessors have been designed with extra scrutiny. AMD, Intel and others use automated theorem proving to verify that division and other operations are correctly implemented in their processors.

2.5 Abstraction

Symbolic representation alone is not enough for addressing the "state-space explosion" problem. The research presented in this dissertation focuses on *abstraction* — an over-approximation of a model that preserves the behavior of the original program.

2.5.1 Abstraction of a transition system

Definition 10. *Given two TSs $M = \langle V, I, T\rangle$ and $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T}\rangle$, a relation $H(V, \hat{V})$ is an* abstraction relation *iff the following conditions hold:*

- *every initial state of M corresponds to an initial state of $\hat{M}$; namely, if $s \models I(V)$, then there exists a state $\hat{s}$ of $\hat{M}$ such that $\hat{s} \models \hat{I}(\hat{V})$ and $s, \hat{s} \models H(V, \hat{V})$;*
- *every transition of M corresponds to a transition of $\hat{M}$; namely, if $s_1, \hat{s}_1 \models H(V, \hat{V})$, and $s_1, s_2' \models T(V, V')$, then there exists a state $\hat{s}_2$ of $\hat{M}$ such that $s_2, \hat{s}_2 \models H(V, \hat{V})$ and $\hat{s}_1, \hat{s}_2' \models \hat{T}(V, V')$.*

If such a relation exists, then $\hat{M}$ is an abstraction *of M, or M* refines *$\hat{M}$ ($M \preceq \hat{M}$).*

Definition 11. *Given the abstraction relation H, the* abstraction function *is defined as $\alpha_H : 2^S \to 2^{\hat{S}}$ and the* concretization function *$\gamma_H : 2^{\hat{S}} \to 2^S$ as follows:*

- *$\alpha_H(Q) = \{\hat{s} \in \hat{S} \mid$ there exists $s \in Q$ s.t. $s, \hat{s} \models H(V, \hat{V})\}$, for every $Q \subseteq S$;*
- *$\gamma_H(\hat{Q}) = \{s \in S \mid$ there exists $\hat{s} \in \hat{Q}$ s.t. $s, \hat{s} \models H(V, \hat{V})\}$, for every $\hat{Q} \subseteq S_{\hat{V}}$.*

γ can be extended to transitions and paths so that:

- $\gamma_H(\hat{t}) = \{t \mid in(t) \in \gamma(in(\hat{t})), out(t) \in \gamma(out(\hat{t}))\}$, for every transition $\hat{t}$ of $\hat{M}$.
- $\gamma_H(\hat{\pi}) = \{\pi \mid \pi[i] \in \gamma(\hat{\pi}[i])$ for every $0 \leq i \leq |\hat{\pi}|\}$, for every path $\hat{\pi}$ of $\hat{M}$.

From the various of techniques developed for abstraction computation we elaborate on those that are relevant to the algorithms developed in this dissertation. The techniques are abstract interpretation, predicate abstraction and predicate abstraction-based counterexample-guided abstraction refinement.

2.5.2 Abstract interpretation

Abstract interpretation [CC77] is a theory of sound approximation of program models. It constructs an abstraction of a program with regards to values from an *abstract domain* by iteratively applying the instructions of a program to abstract values until the fixpoint is not reached.

Formally, for a program $P = \langle U, G \rangle$, where U is the universe and G is a program graph, the (concrete) semantics of a program is given by the pair $\langle A, \tau \rangle$, where:

- A is the set of *assertions* of the program, where each assertion $p \in A$ is a predicate over U; $A(\Rightarrow, false, true, \vee, \wedge)$ is a complete Boolean lattice;
- $\tau : L \rightarrow (A \rightarrow A)$ is the predicate transformer.

An *abstract interpretation* is a pair $\langle \hat{A}, t \rangle$, where $\hat{A}$ is a complete lattice $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, and $t : L \rightarrow (\hat{A} \rightarrow \hat{A})$ is a predicate transformer. Note that $\langle A, \tau \rangle$ is a particular abstract interpretation called the *concrete interpretation*. In the following, we assume that for every command $c \in L$, the function $t(c)$ is monotone (which is the case for all natural predicate transformers). Given a predicate transformer t, the function $\tilde{t} : L^* \rightarrow (\hat{A} \rightarrow \hat{A})$ is recursively defined as follows:

$$\tilde{t}(p)(\phi) = \begin{cases} \phi & \text{if } p \text{ is empty} \\ \tilde{t}(e)(t(q)(\phi)) & \text{if } p = q; e \text{ for some } q \in L,\ e \in L^*. \end{cases}$$

2.5.3 Predicate abstraction

Predicate abstraction [GS97, CU98] is one of the most popular and widely applied methods used in model checking for systematic abstraction of programs.

It abstracts data by only keeping track of certain predicates on the states. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state model that is an abstraction of the original system with respect to a set of predicates.

Predicate abstraction relation

Given a TS $M = \langle V, I, T \rangle$, let Π be a set of predicates and $\hat{v}_p$ an abstract variable for every predicate $p \in \Pi$. The set of abstract variables is the set $\hat{V}_\Pi = \{\hat{v}_p\}_{p \in \Pi}$. The abstraction relation for predicate abstraction is defined as follows:

$$H_\Pi(V, \hat{V}_\Pi) = \bigwedge_{p \in \Pi} \hat{v}_p \longleftrightarrow p(V)$$

2.5.4 Abstraction-refinement loop

The system abstraction, even if computed precisely, may contain unrealistic behaviors: i.e., some traces of the abstract system cannot be simulated on the concrete system. If, after the verification of a property, it is found that a counterexample is not simulatable, it is considered *spurious*. The presence of such non-realistic traces requires one to remove them in order to be able to conclude if the property holds or not. The CounterExample-guided Abstraction Refinement (CEGAR) [CGJ+00, Kur95] is a technique that automates this process.

In case of predicate abstraction, CEGAR consists of a loop that maintains a set Π of predicates incremented at every iteration.

Each iteration of the CEGAR-loop has 4 basic steps:

- *abstraction*, where it builds an abstract system $\hat{M}$ according to a given set of predicates Π;

- *verification* (or model checking), where it checks if all the executions of $\hat{M}$ satisfy the property; if the property is correct in the abstract system, it is concluded that it is also true in the concrete system;

- *simulation*: if the verification produces a counterexample, the simulation checks if it is spurious, by simulating it in the concrete program; if the counterexample is simulatable, it is reported as a real counterexample;

- *refinement*: if the simulation establishes that the counterexample is spurious, the refinement is performed.

The refinement step strongly depends on the type of the spurious behavior the counterexample reveals. In Chapter 3, we detail on the types of spurious behaviors, refinement approaches and propose an algorithm that optimizes the number of required CEGAR-loop iterations.

2.6 Implementations of verification techniques

Application of a verification technique to industrial problems is a difficult task that require a major engineering effort. What usually helps is a strong commercial interest behind additional product quality assurance obtained with mathematically-provided guarantee of product correctness. A good example of it are tools that were developed to verify Microsoft Windows device drivers (e.g. SLAM [BCLR04]).

Several successful verification tool sets are described below.

SPIN

One of the most successful tools in the history of formal methods is SPIN [Hol03]. SPIN is a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion. It was written by Gerard J. Holzmann and others in the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. It found its major application in verification of mission-critical software at NASA. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

An explicit-state model checking algorithm behind SPIN requires finite-state transition systems to operate. Systems to be verified are described in Promela (Process Meta Language), which supports modeling of asynchronous distributed algorithms as non-deterministic automata (SPIN stands for “Simple Promela Interpreter”). Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm.

The work in this thesis tackles the problem of software verification in orthogonal way. Instead of focusing on optimization of the explicit reasoning over the state space of a program, we aim to build a precise and concise abstraction for it and, thus, to obtain a smaller model and to gain in time required for a verification step. However, since our techniques produce finite state transition systems, they can serve as a step before application of SPIN.

SLAM

The SLAM project [BCLR04] was started by Microsoft in response to an urgent need of improving the correctness of 3-rd party device drivers in Windows. The failures in those lead to notorious BSODs (Blue Screen of Death) — unrecoverable system errors in Windows family of operating systems. SLAM is a set of tools for checking that software (namely, device drivers) satisfies critical behavioral properties of the interfaces it uses. Static Driver Verifier is a tool in the Windows Driver Development Kit that uses the SLAM verification engine. It aids software engineers in designing interfaces and software that ensure reliable and correct functioning.

The SLAM verification engine operates on boolean programs — special kind of abstract program representation. Those are obtained by an abstraction procedure that focuses on correct handling of device drivers C code and, in particular, on correctness of communication via driver interfaces [BMMR01].

We target the analysis of general purpose software and therefore we cannot compare directly with the tools in the SLAM project. However there is another Microsoft Research tool, TERMINATOR, that we do compare with. TERMINATOR inherited from SLAM its focus on device drivers correctness and we tackle the same problem of program termination. The details of the comparison are in Chapter 5.

Bandera

Bandera is a tool set for model checking of concurrent Java software [CDH$^+$00]. The Bandera project addresses one of the major obstacles in the path of practical finite-state verification of software — generation of finite input models. Tools like SMV and SPIN accept a description of a finite-state transition system as input. Bandera bridges the semantic gap between a non-finite-state software system expressed as source code and SMV/SPIN input languages. by providing sophisticated program analysis, abstraction, and transformation techniques.

The goal of the Bandera project is to integrate existing programming language processing techniques with newly developed techniques to provide automated support for the extraction of safe, compact, finite-state models that are suitable for verification from Java source code. The algorithms presented in this thesis share the motivation — provide automatic generation of finite models from programs. In particular, loop summarization allows scalable handling of relatively large code bases. We did not found an algorithm with an approach similar to ours in the set of Bandera tools, thus, it can be added to enrich the current bundle. However, it should be noted that we focus on summarization of sequential programs and application to concurrent software should be further researched.

SatAbs

The experimental goal of this thesis is to develop tools that enable verification of security properties in real programs. SatAbs, a model checker for ANSI-C programs [CKSY05a], is used as a platform for implementations of new techniques within a CEGAR loop (described in details in Chapter 3). SatAbs features a predicate abstraction refinement loop where SAT-Solver is used as a decision procedure for abstraction, simulation and refinement [CKSY04]. It provides both existential [CGL94] and over-approximated [BPR03] abstractions, allows model checking not only sequential but also concurrent programs with asynchronous interleaving semantics [CKS05].

Apart from faster computation, the usage of SAT-Solver in SatAbs also gives the model checker an ability of precise reasoning on bit-vectors, common constructs in hardware and software applications [CKS06]. As a result program semantics of the ANSI-C standard is handled accurately, i.e. SatAbs treats soundly and without over-approximations bit-vector and array overflows, pointers arithmetic, union operations and other ANSI-C constructs. That makes SatAbs a valuable tool for code security analysis.

CBMC

Another tool partially employed for experimental evaluation in this thesis is CBMC — bounded model checker for ANSI-C and C++ programs [CKL04]. The verification is performed by unwinding loops in a program and passing the resulting equation to a decision procedure. As well as SatAbs it favors precise handling of C semantics by relying on bit-precise decision procedures.

Our tool Loopfrog presented in Chapter 4 makes use of symbolic execution engine of CBMC. However we do not unwind loops but over-approximate them with abstract summaries. LOOPFROG can be applied to reason soundly about programs, on which CBMC fails to unwind a loop and thus cannot terminate the verification process.

Chapter 3

Synergy of Precise and Fast Abstraction

Absolute truth exists.
But it is absolutely useless.
<from philosophical debates>

This chapter presents a CEGAR-based technique that controls the number of abstraction-refinement iterations and reduces the verification time by interleaving precise (but slow) and approximated (but fast) abstractions. The abstraction is first computed with a high level of approximation exploiting the weakest precondition of the predicates. Then, during the refinement step, the technique uses the SAT-based quantifier elimination in order to compute the precise abstraction over the restricted subset of predicates relevant to the discovered counter-example. We also show how the precise computation can be further heuristically restricted in order to avoid possible exponential blow up caused by increase in the number of predicates.

Overall, the new technique manages the verification complexity by using the precise abstraction on demand and locally. The advantage is that the expensive abstraction is only used on a small portion of the program, yet the higher quality of abstraction refinement is sufficient to reduce the number of refinement iterations, thus improving the overall performance of verification.

3.1 Introduction

Predicate abstraction, when combined with reachability analysis and an automated abstraction refinement mechanism, is an effective model checking strategy. The CEGAR-based verification using predicate abstraction consists of constructing and evaluating a finite-state system that is an abstract model of the original system with respect to a set of predicates.

The abstract model is a conservative over-approximation of the original program with respect to the set of given predicates. Thus, if the property holds on the abstract model, it also holds on the original program. The drawback of the conservative abstraction is that it can introduce unrealistic behaviors, i.e., model checking phase of CEGAR-loop can discover spurious counterexamples — valid for abstract model, but not realizable on a concrete system. Such spurious behaviors should be removed by adjusting the set of predicates in a way that eliminates the given counterexample. Therefore, the overall efficiency of verification is highly dependent on the efficiency of the abstraction and refinement procedures.

We distinguish between *precise* abstraction and *approximated* abstraction (as also done, for example, in [CGJ$^+$00, DD01, JM05]): a precise abstraction is minimal in the sense that it contains only those transitions that correspond to some transition in the concrete model; instead, an approximated abstraction is a further over-approximation of the minimal abstract model so that the transition relation is relaxed. In this work we refer to the latter simply as approximation.

Approximation techniques are important because they allow a less expensive (as compared to precise abstraction) computation of the abstract transition relation. Cartesian abstraction [BPR03], for example, loses every relationship among predicates, but has been successfully used to verify large programs, such as operating system device drivers. However, abstraction approximations add spurious behaviors in addition to the spurious counterexamples resulting from precise abstraction. In order to rule out this kind of "impurity", the approximation must be refined without changing the set of predicates and focusing only on the spurious transitions caused by the approximation [DD01]. This procedure on its own might become very costly and does not scale to verification of large programs.

When refining the abstract model, we distinguish between two types of spurious behavior (as also done in [CGK$^+$02]). 1) *Spurious path* is due to the over-approximating nature of the precise abstraction: states are merged together so that some resulting paths cannot be simulated on the concrete system. This happens when the set of predicates is not sufficient to capture the relevant behaviors of the concrete system. 2) *Spurious transitions* are abstract transitions which do not have corresponding concrete transitions. This happens when the set of transitions is over-approximated too much while abstraction is

```
    void main() {
    int x=*;
    int y;
l0: y=x+1;
l1: if (x<0)
l2: if (!(x<y))
l3: assert(0);
l4: }
```

(a)

```
void main() {
int x = *;
int y;
int z;
y=x+1;
z=y-1;
if (x<0||y>0||z>0){
x++;
z++;
if (y<z && z>x)
assert(0); } }
```

(b)

Figure 3.1. Sample program for which the approximated abstraction causes spurious paths.

computed. By definition, spurious transitions cannot appear in the most precise abstraction and are caused by using the approximation techniques. Clearly, the efficiency of the approximated abstraction depends on a tradeoff between time spent in computing the abstraction and refining spurious transitions.

In order to illustrate the abstraction approximation and its refinement procedures, consider the example of Figure 3.1(a). The variable x is assigned non-deterministically with an unknown value "$*$". The property we verify is the reachability of line l3. It never can be reached since the condition !(x<y) at line l2 never holds (together with the guard x<0 at line l1, which is sufficient to avoid integer overflow). Thus if in the abstract program there is a path leading to the assertion, then it is spurious. The predicates x<0 and x<y are sufficient to prove the property. However, approximate methods like Cartesian abstraction cannot prove it because they cannot infer that after the assignment y=x+1, the condition (!(x<0) || !(x<y)) is true. Thus, most model checkers that use such abstractions refine the transition relation by adding a constraint that removes the spurious transition.

To demonstrate the difference in performance between precise and approximated abstractions, let us extend the previous example in order to have more spurious behaviors. The program of Figure 3.1(b) has one more variable and a slightly more complex control flow graph. As before the assertion is not reachable, and all abstract counterexamples are spurious. Though, if we consider the predicates in the guards of the program, an approximated abstraction may produce many spurious behaviors. Table 3.1 reports the verification results obtained with the SatAbs model checker [CKSY04], by running approximated

	Total	Abs	MC	Ref	Iter
Approximated abstraction [DD01]	5.817	0.063	2.659	2.112	42
Approximated abstraction [JKSC05]	1.469	0.046	0.501	0.617	12
Precise abstraction [CKSY04]	3.591	3.478	0.076	0.01	2
New approach	0.467	0.039	0.161	0.189	4

Table 3.1. Verification results on the example presented in Figure 3.1(b). *Total, Abs, MC, Ref* refer to the time, in seconds, for total verification, abstraction, model checking and refinement respectively; *Iter* refers to the number of iterations of the abstraction-refinement loop.

and precise abstractions. The final number of predicates is in all cases 10. The approximated abstraction spends most of time in refining the transition relation (Ref). Since it runs for 12 iterations (or even 42 in case when we used the refinement procedure of [DD01]), also the time for the verification (MC) is not negligible. On the contrary, the precise abstraction takes only 2 iterations to terminate (the first refinement is necessary to add a sufficient set of predicates). Nevertheless, the amount of time spent in computing the abstraction is too high for such example.

A low number of refinement iterations is fundamental for the success of the CEGAR loop, especially when applied to industrial benchmarks: in fact, when the system is complex, the number of predicates required to verify the property becomes high, and the time spent in the reachability (model checking) procedure grows exponentially. For this reason, it is of paramount importance to avoid as many redundant iterations as possible: even a single saved iteration can result into a huge saving in time for large systems.

This chapter presents a new technique that controls the number of iterations in CEGAR loop and reduces the verification time by interleaving precise (but slow) and approximated (but fast) abstractions. The abstraction is first computed with a high level of approximation. Then, during the refinement step, the precise abstraction is applied to a limited subset of transitions, related to detected spurious behavior.

Notably, the method is independent of any particular technique used to define either abstraction procedure.

The difficulty that one would experience in computing the precise abstraction of the whole program is avoided by exploiting the localized abstraction: as in static analysis [NNH99], in most model checkers (such as SLAM [BMMR01], BLAST [HJMS02], SATABS [CKSY04], F-Soft [JIGG05]) the abstract model keeps the control flow graph of the original program and has a different abstract transition relation for each location of the control-flow graph[1]. This way, during the refinement step, we add the constraints built with a precise abstraction only to relevant transition relations, affecting only those parts of the system that caused the spurious counterexample.

In order to illustrate the immediate advantages of our approach, consider the fourth line of Table 3.1 that is based on the implementation of our technique. Our approach is able to avoid both a high number of iterations and an expensive abstraction, resulting in an optimized verification time.

The rest of the Chapter is organized as follows: Section 3.2 defines precise and fast abstractions; Section 3.3 describes the algorithm to combine them in a CEGAR-loop. Next, Section 3.4 discusses the experimental support for the new algorithm including the modification with the threshold on precise computation. Finally, Section 3.5 summarizes and relates presented contribution to the existing work.

3.2 Precise abstraction vs. fast abstraction

We deal with a program modeled as a transition system (TS) $M = \langle V, I, T \rangle$ (Definition 2) and rely on a notion of abstraction for a transition system as introduced in Section 2.5. We also make use of earlier defined predicate abstraction (Section 2.5.3) and CEGAR-loop (Section 2.5.4).

3.2.1 Precise abstraction

Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$ of M is said to be *precise* when every abstract initial state and transition of $\hat{M}$ corresponds respectively to a concrete initial state and transition of M, i.e., for every abstract transition, every corresponding concrete transition can occur. Given the abstraction relation H, $\hat{M}$ can be obtained as:

- $\hat{I}_H(\hat{V}) = \exists V (I(V) \wedge H(V, \hat{V}))$,
- $\hat{T}_H(\hat{V}, \hat{V}') = \exists V \exists V' (T(V, V') \wedge H(V, \hat{V}) \wedge H(V', \hat{V}'))$

The precise abstraction is also called *minimal* or *existential* or *exact* or *eager* abstraction [CGL94].

[1]Localized abstraction is further investigated in [HJMS02, HJMM04].

Precise predicate abstraction

The minimal predicate abstraction is the TS $\hat{M} = \langle \hat{V}_\Pi, \hat{I}_\Pi, \hat{T}_\Pi \rangle$, where:

- $\hat{I}_\Pi(\hat{V}_\Pi) = \exists V(I(V) \wedge \bigwedge_{p \in \Pi} \hat{v}_p \longleftrightarrow p(V))$
- $\hat{T}_\Pi(\hat{V}_\Pi, \hat{V}'_\Pi) = \exists V \exists V'(T(V,V') \wedge \bigwedge_{p \in \Pi}(\hat{v}_p \longleftrightarrow p(V) \wedge \hat{v}'_p \longleftrightarrow p(V')))$.

Quantifier elimination

While attempting to model check the precise abstract TS, one faces the need to remove the quantifiers from the above defined abstract transition relation. In general, given a transition relation T and a set of predicates P, *to compute* $\hat{T}_P$ means to find a quantifier-free formula that is equivalent to $\hat{T}_P$.

Example 3.1. *Consider the program of Figure 2.1(a). It can be represented by the TS* $M = \langle V, I, T \rangle$*, where*

- $V := \{x, y, pc\}$*, where pc is the program counter;*
- $I := (pc = l_0)$*;*
- $$\begin{aligned} T := \ & (pc = l_0) \rightarrow (pc' = l_1 \wedge y' = x + 1 \wedge x' = x) \wedge \\ & (pc = l_1 \wedge x < 0) \rightarrow (pc' = l_2 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_1 \wedge !x < 0) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_2 \wedge !x < y) \rightarrow (pc' = l_3 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_2 \wedge x < y) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_3) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \end{aligned}$$

Now consider the predicates $P_1 := (x < 0)$ *and* $P_2 := (x < y)$*. Let the abstract variables* $\hat{v}_1$ *and* $\hat{v}_2$ *correspond respectively to* P_1 *and* P_2*. We do not abstract the program counter. The precise abstract transition relation results to be equivalent to*

- $$\begin{aligned} \hat{T}_P \equiv \ & (pc = l_0 \wedge \hat{v}_1) \rightarrow (pc' = l_1 \wedge !\hat{v}'_2) \wedge \\ & (pc = l_0 \wedge !\hat{v}_1) \rightarrow (pc' = l_1) \wedge \\ & (pc = l_1 \wedge \hat{v}_1) \rightarrow (pc' = l_2 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_1 \wedge !\hat{v}_1) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_2 \wedge !\hat{v}_2) \rightarrow (pc' = l_3 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_2 \wedge \hat{v}_2) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_3) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \end{aligned}$$

In hardware and software verification, different techniques have been conceived to compute $\hat{T}_P$. In symbolic model checking [BCM+90] of finite state machines, the existential quantification can be removed either by a Shannon expansion technique when using BDDs [Bry86] or by SAT techniques when using CNF [McM02]. In software model checking, the problem is exacerbated by the fact that the concrete transition relation may contain first-order terms. The abstract transition relation can be obtained by enumerating the abstract states, and checking if, for each pair of states, there exists an abstract transition. As it is done by most software model checkers, this requires an exponential number of calls to a theorem prover [DDP99, BMMR01]. In [CKSY04] a SAT-Solver is exploited for this purpose, the technique is known as `SATQE` — SAT-based quantifier elimination.

3.2.2 Approximated abstraction

Precise abstractions are very expensive to compute because of the existential quantification operations. Thus, in practice, model checkers use approximations to trade-off precision with complexity. Formally:

Definition 12. *Given $M_H = \langle V, I_H, T_H \rangle$ and $\tilde{M} = \langle V, \tilde{I}, \tilde{T} \rangle$, $\tilde{M}$ is an* approximation *of M_H ($M_H \precsim \tilde{M}$) iff the following formulas are valid:*

- *$I_H \implies \tilde{I}$, i.e., every initial state of the minimal abstraction is an initial state in the approximation;*
- *$T_H \implies \tilde{T}$, i.e., every transition of the minimal abstraction is a transition in the approximation.*

Intuitively, $\tilde{M}$ has more initial states and transitions than M_H. Note that an approximation is also an abstraction namely, if $M_H \precsim \tilde{M}$, then $M_H \preceq \tilde{M}$. However, the set of predicates is not affected, in the sense that $\tilde{M}$ and M_H have the same abstract variables.

Approximation for predicate abstraction

Many approximation techniques have been developed both in hardware and software verification. Their aim is to alleviate the computation of $\hat{T}_\Pi$. The easiest way is to reduce the scope of quantifiers. This can be done with *early quantification* [CGL94], by pushing quantifiers in front of predicates. *Predicate partitioning* [JKSC05] approximates $\hat{T}_\Pi$ by taking the conjunction of its projections over subsets of predicates. This technique is pushed to its limit by

Cartesian abstraction [BPR03] that, given a set of states Q, approximates transition relation with the product of the projections on each variable. This way, the approximated abstraction ignores every relation among predicates.

3.2.3 Spurious behaviors

Spuriousity, which is introduced during abstraction computation, should be automatically removed during the refinement phase of CEGAR-loop. The refinement step strongly depends on the type of the spurious behavior the counterexample reveals. Two types are possible:

Spurious transitions

Spurious transitions are transitions that satisfy the abstract transition relation, but not the concrete one.

Definition 13 (Spurious transition). *Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$, and a transition $\hat{t}$ of $\hat{M}$, $\hat{t}$ is a spurious transition iff $\hat{t} \models \hat{T}$ and $t \not\models T$ for every $t \in \gamma(\hat{t})$.*

In order to refine an approximation that contains a spurious transition, a new transition relation is obtained by adding a constraint in conjunction to the old abstract transition relation. As a result, the spurious counterexample is ruled out. Different techniques use as such a constraint either the exact encoding of the spurious transition [DD01], or the UNSAT core produced by the SAT solver when checking if the transition is spurious [JKSC05], or an interpolant between the exact abstraction and the current approximated abstraction [JM05].

Note that if the abstraction is precise, there is no need for spurious transition refinement, because this type of spurious behavior can not happen in minimal abstraction by definition. However the existence of spurious paths is still possible.

Spurious path

If there are no spurious transitions (which is the case for precise abstraction), but the counterexample is spurious, then there is a spurious path. Spurious paths are sequences of transitions that satisfy the abstract transition relation, but not the concrete one.

Definition 14 (Spurious path). *Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$, and a sequence $\hat{\pi}$ of transitions of $\hat{M}$, $\hat{\pi}$ is a spurious path iff $\hat{\pi} \models \hat{T}$ and $\pi \not\models T$ for every $\pi \in \gamma(\hat{\pi})$.*

In order to refine the abstraction and remove a spurious path, refinement procedures need to add more predicates to the abstraction. There are different techniques to discover the new set of predicates, either based on weakest precondition [BR02], interpolation [HJMM04], or UNSAT core [GS05].

3.3 Combining fast and precise abstractions

Now we define a new algorithm, which combines fast and precise abstraction. The algorithm implements the standard CEGAR-loop as described in Section 2.5.4. Each iteration of the CEGAR-loop is composed of an abstraction step, a model checking step, a simulation step and, finally, a refinement step.

First we present here the high-level overview of the combined algorithm and then we provide the description of the specifics of the new refinement procedures. For simplicity, the algorithm is initially explained with regard to a monolithic transition relation. In Section 3.3.3 it is extended to the case where a transition relation is defined for every location of the program.

The algorithm is parametrized by a number of subroutines that take care of the abstraction and refinement steps. In particular, the algorithm contains the following procedures:

- `FastAbstraction`: given a set of predicates Π and a concrete transition relation T, it computes an over-approximation of $\hat{T}_\Pi$.
- `PreciseAbstraction`: given a set of predicates Π and a concrete transition relation T, it computes the minimal abstraction $\hat{T}_\Pi$.
- `SpuriousTransition`: given a path π, it returns a function σ_{ST} that maps every transition t in π to a set of predicates P, s.t., $P \subseteq \Pi$ and $t \not\models \hat{T}_P$.
- `SpuriousPath`: given a path π, it returns a function σ_{SP} that maps every transition t in π to a set of predicates P, s.t. $\pi \not\models \hat{T}_P$. Note that P may contain new and old predicates.

Also, note that, depending on the detected counterexample pi and P, both σ_{ST} and σ_{SP} may or may not be defined.

Algorithm 1 shows how the `FastAbstraction` and `PreciseAbstraction` are combined. It first computes the approximated abstraction (line 4). When a spurious counterexample is encountered as a result of the model checking (line 6), the spurious transitions are removed by using the precise abstraction technique (line 12) with the predicates returned by `SpuriousTransition` (line 9). If no spurious transitions are found, the spurious path is removed by using the precise abstraction technique (line 21) with the predicates returned by `SpuriousPath` (line 16).

3.3.1 Refining spurious transitions

Suppose some transitions $t_1, ..., t_n$ of the counterexample π found by `ModelCheck` are spurious. This means that the function σ_{ST} returned by `SpuriousTransition` maps those transitions to some non-empty set of predicates. The clustering of predicates Γ is defined as $\{\sigma_{ST}(t_i)\}_{1 \le i \le n}$ (i.e., Γ contains the set of predicates $\sigma_{ST}(t_i)$ for every transition in the spurious counterexample). The spurious transition refinement procedure proceeds as follows. For each cluster, $P \in \Gamma$, the refinement algorithm computes $\hat{T}_P$, which is a precise computation of the abstract transition relation projected on the predicates of the cluster. In order to rule out every spurious transition among $t_1, ..., t_n$, the refinement algorithm updates the abstract transition relation α as follows: for each set of predicates, it adds the precise abstraction of the transition relation projected on those predicates. The updated abstract transition relation becomes:

$$\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$$

Note that, in general, every cluster, P, is a subset of the global set of predicates, Π. This means that each constraint $\hat{T}_P$ is an over-approximation of the precise abstraction computed over Π. Neverthless $\hat{T}_P$ is precise with regards to the predicates P, in the sense, that *it removes all the unrealistic abstract transitions that can be defined by those predicates*.

The following theorem states the soundness of this refinement step.

Theorem 2. *For every spurious transition t_i, $1 \le i \le n$, $t_i \not\models \alpha'$.*

Proof. By definition of the function σ_{ST}, it maps every transition t_i, $1 \le i \le n$, to a set of predicates P such that $t_i \not\models \hat{T}_P$. Since $\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$, then $t_i \not\models \alpha'$. □

Algorithm 1: A new abstraction-refinement algorithm combining fast and precise abstractions. α — abstract transition relation (symbolic representation); T — concrete transition relation; Π — predicate set; π — counterexample; C — constraints added to abstract transition relation on refinement step.

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    $\Pi$ = InitialPredicates($F$,$T$);
    $\alpha$ = FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi$ = ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST}$ = SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
                    $\alpha = \alpha \wedge C$;
                end foreach
            else
                $\sigma_{SP}$ = SpuriousPath($\pi$);
                if $\sigma_{SP} = \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $\Pi = \Pi \cup \sigma_{SP}(t)$;
                        $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
                        $\alpha = \alpha \wedge C$;
                    end foreach
                end if
            end if
        end if
    end while
end
```

Note that the proof relies on the soundness of a particular `SpuriousTransition` and `PreciseAbstraction` techniques. For a correctly returned by `SpuriousTransition` set of predicates, `PreciseAbstraction` guarantees to produce a constraints sufficiently strong to remove a spurious transition between the states that predicates are related to.

3.3.2 Refining spurious paths

The cluster-based approach described above is adopted to the removal of the spurious path. The technique uses `SpuriousPath` to produce the set of predicates that are sufficient to rule out the spurious counterexample. The set of predicates generated by the standard predicate-discovery techniques (described in Section 2.5.3) includes both current predicates and new predicates, that together rule out the spurious counterexample. Our technique considers this set of old and new predicates as a new cluster.

Suppose the path $t_1, ..., t_n$ to be spurious. This means that the function σ_{SP} returned by `SpuriousPath` maps each t_i to some non-empty set of predicates. The clustering of predicates Γ is defined as $\{\sigma_{SP}(t_i)\}_{1 \leq i \leq n}$ (i.e., Γ contains the set of predicates $\sigma_{SP}(t_i)$ for every transition in the spurious counterexample). For each set of predicates, we add a precise computation of the abstract relation with regards to those predicates. The computation of the updated abstract transition relation α is identical to spurious transition case, i.e.,

$$\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$$

Note that this time, unlike the case of spurious transitions, the clusters involve new predicates.

The advantage of this refinement procedure over classical path-based refinement techniques is that not only does it use new predicates to remove spurious paths, but it also makes use of precise components to make sure that no spurious transitions are created as a result of the refinement step. In the long run, it helps to reduce the number of the CEGAR-loop iterations as well.

By definition, the set of predicates produced by `SpuriousPath` is sufficient to remove the spurious counterexample only if the precise abstraction is used. In fact, spurious transitions over such predicates (possibly created by the approximation abstraction) might create the same spurious counterexample. The proposed technique guarantees that this does not happen. This is achieved by using the precise component $\hat{T}_P$.

The following theorem states the soundness of this refinement step.

Theorem 3. *For every spurious path π, $\pi \not\models \alpha'$.*

Proof. By definition of the function σ_{SP}, it maps every transition t of π, to a set of predicates P such that $\pi \not\models \hat{T}_{\sigma_{SP}(t)}$, i.e., precise abstraction, computed for any cluster P, rules out a spurious path π. Since abstract transition relation is updated with a constraint for every cluster P, i.e., $\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$, then $\pi \not\models \alpha'$. □

Note that the proof relies on the soundness of a particular `SpuriousPath` technique.

3.3.3 Localized abstraction

The algorithm shown in Algorithm 1 was defined for a monolithic transition relation. When the set of predicates returned by the `SpuriousTransition` or `SpuriousPath` procedures covers the whole set Π of current predicates, the constraint that `MixCegarLoop` adds to the abstract transition corresponds exactly to the precise abstraction. This way, the abstraction refinement becomes as expensive as `PreciseAbstraction`. This disadvantage is limited by localizing the abstraction to some parts of the program. Some software model checkers (e.g., BLAST [HJMS02] and SATABS [CKSY04]) use the control flow graph as a partitioning of the transition relation to implement such localization. During the abstraction refinement, they keep a set of predicates and an abstract transition relation for each program location, and perform the abstraction for each local transition relation separately.

The algorithm implements the localized procedure as part of the CEGAR loop as shown in Algorithm 2. It treats the system M as a set of concrete transition relations, one for every location of the control-flow graph. For each transition relation T, it computes an abstract transition relation $\alpha(T)$ (line 4); when a spurious counterexample is encountered as a result of the model checking (line 6), spurious transitions and paths are removed by using the precise abstraction technique (line 13 and 23). The difference from the monolithic case (presented earlier in this section) is that in the localized version, every transition t of the spurious counterexample π is associated with a particular abstract transition relation, denoted $\tau(t)$. Thus, when the refinement step of the algorithm has to add a new constraint, it changes only the transition relation corresponding to either the spurious transition (as part of the spurious

Algorithm 2: An algorithm that implements a combination of fast and precise abstractions together with localized abstraction.

```
MixCegarLoop(TransitionSystem M, Property F)
begin
    foreach T in M do $\Pi(T) =$ InitialPredicates(F,T);
    foreach T in M do $\alpha(T) =$ FastAbstraction(T,$\Pi$);
    while not TIMEOUT do
        $\pi =$ ModelCheck($\alpha$,F);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma =$ SpuriousTransition($\pi$);
            if $\sigma \neq \emptyset$ then
                foreach $t \in \pi$ do
                    $T = \tau(t)$;
                    $C =$ PreciseAbstraction(T,$\sigma(t)$);
                    $\alpha(T) = \alpha(T) \wedge C$;
                end foreach
            else
                $\sigma_{SP} =$ SpuriousPath($\pi$);
                if $\sigma_{SP} = \emptyset$ then return INCORRECT;
                else
                    foreach $t \in \pi$ do
                        $T = \tau(t)$;
                        $\Pi(T) = \Pi(T) \cup \sigma_{SP}(t)$;
                        $C =$ PreciseAbstraction(T,$\sigma_{SP}(t)$);
                        $\alpha(T) = \alpha(T) \wedge C$;
                    end foreach
                end if
            end if
        end if
    end while
end
```

transition refinement step, lines 9-14) or to each transition of the spurious path (as part of the spurious path refinement step, lines 17-24).

By exploiting the localized-abstraction framework, the algorithm reduces the abstraction computation to the parts of the system that are relevant to the property and keeps the approximated abstraction in all parts of the program that are irrelevant to prove the property.

3.4 Experimental evaluation

We performed a thorough evaluation comparing the new technique with the purely precise and imprecise counterparts. Our tests with various real life benchmarks show a systematic advantage of our approach over both precise and imprecise techniques reaching up to 90% improvement in time.

We implemented the proposed algorithm in the framework of software model checking. We used the SatAbs [CKSY04] model checker as a platform for our experiments. As described in Section 3.3, the new CEGAR loop uses four subroutines. We experimented with the following techniques implemented in SatAbs:

- for `FastAbstraction`, we used a fast abstraction technique based on the computation of the weakest precondition; it assigns to the next predicate its weakest precondition if this is a current predicate; it does not allow a general Boolean combination of predicate variables;

- for `PreciseAbstraction`, we used a precise abstraction based on the enumeration of possible transitions by means of a SAT solver: we force the SAT solver to find all the solutions of the quantifier-elimination problem by iteratively adding the negation of previous assignments as clauses [CKSY04];

- for `SpuriousTransition`, we used the SAT-based technique of [JKSC05][2]; this calls a SAT solver to check if a transition is spurious; if the transition is not realistic, it inspects the UNSAT proof to find the relevant predicates;

- for `SpuriousPath`, we used a technique based on weakest precondition; it computes the weakest preconditions of the current predicates along the transitions of the spurious path; it uses these expressions to produce a set

[2]We also experimented with a direct implementation of technique [DD01], but it reached 200 CEGAR iterations even on the small examples.

of current and new predicates that are sufficient to rule out the spurious path.

The SAT solver used by `PreciseAbstraction` and `SpuriousTransition` was MiniSAT [ES03].

We implemented the new algorithm and enhanced SatAbs with two new procedures: the first (we will refer to it as `NewST`) affects how the abstraction is refined in the case of spurious transitions, as described in Section 3.3.1; the second (`NewSP`) refines the abstraction in the case of spurious paths, as described in Section 3.3.2.

We compared the new algorithm with the abstraction-refinement loop based on the pure fast abstraction (referred as `WP`) and the pure precise abstraction (referred as `SATQE`) using the standard SatAbs implementations of latter techniques. The new algorithm was evaluated with either `NewSP` or `NewST` or both together. Thus, in case `NewSP` was not used, the default refinement of SatAbs based on fast abstraction was used.

Note that the focus of the experimental evaluation was on effectiveness of abstraction-refinement approaches. Thus, we maintained the same tool framework and we did not change orthogonal techniques such as predicate discovery.

We ran the experiments on an AMD Dual-Core Opteron 2212 machine with 2GHz CPU and Ubuntu 7.04. The techniques were evaluated on the sets of ANSI-C programs as benchmarks with different assertions in it. For every experiment, we verified one property at a time.[3]

3.4.1 Client-server updating mechanism benchmark

We first compared the analyzed techniques on the C implementation of a client-server mobile agent system [BDNL02, BSBA07], where updates from a central update server are pushed to several clients in form of mobile agents. This enables keeping the client software up to date, without forcing the client to poll the update server.

This benchmark can be parametrized in number of simultaneously executed clients. The increase of the latter parameter results in growth of number of predicates required to verify the complete system. This example is particularly interesting because the fast abstraction produces a number of spurious transitions exponential in the number of predicates.

[3]We observed that verifying several assertions at the same time may affect the comparison in a unreliable way, since the counterexample produced by the model checker may vary according to different abstract models. This way, in the same iteration we might obtain different predicates which might close the CEGAR loop in a different number of iterations.

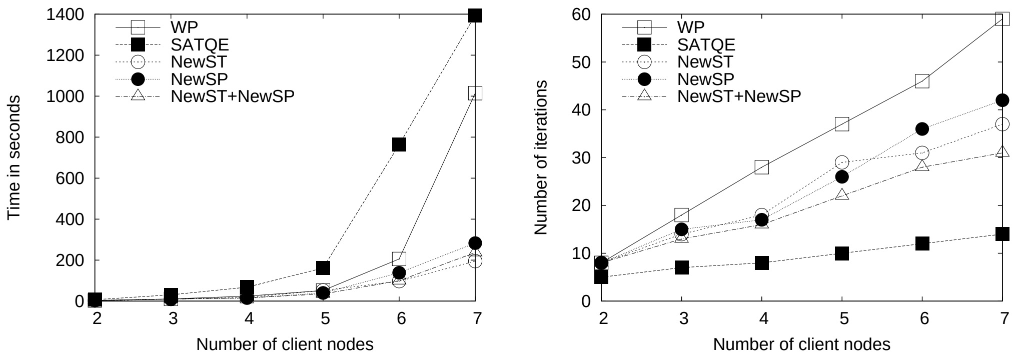


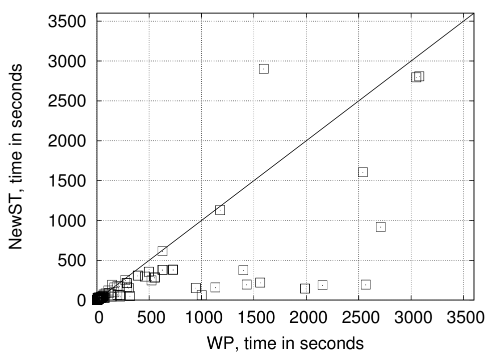
Figure 3.2. Client-server updating mechanism benchmark: total running time in seconds (left) and number of iterations (right) plotted against the number of clients.

The results are reported in Figure 3.2. The performance of the weakest-precondition-based (WP) and the SAT-based abstractions (SATQE) is comparable. Notably, NewST separately and in combination with NewSP is much more efficient than either WP or SATQE. WP and NewSP are sensitive to a number of spurious transitions and, due to the nature of the example, grow exponentially with the size of the model. NewST efficiently removes spurious transitions and significantly reduces the number of iterations. In Figure 3.2 (right) we note that the new technique as expected has a balanced number of iterations between WP and SATQE. This produces an evident saving in time (as shown in Figure 3.2 left) comparing to either WP (up to factor of 5) and to SATQE (up to factor of 7).

3.4.2 Benchmark test suite from Ku et al.

Next, we evaluated the techniques on the benchmark set proposed by Ku et al. in [KHCL07]. For this benchmark set, the authors collected a large number of existing C programs with known buffer-overflow bugs and their fixed versions. The test suite includes applications such as Sendmail, Apache HTTP server, Samba; though, the original programs were stripped down by substituting libraries with stubs. The benchmark set contains 568[4] test cases, of which 261 are fixed versions of the programs.

[4]We reported to the benchmark authors that 17 test cases are incorrect, 31 test cases do not pass correctly through our front-end, thus only 520 test cases were used.



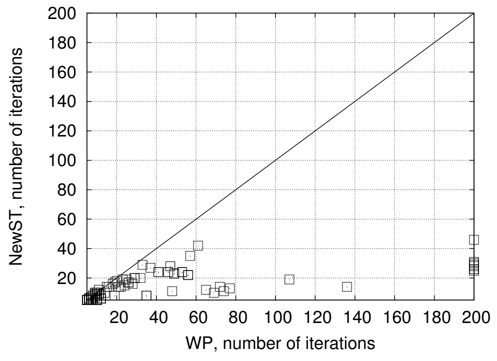


Figure 3.3. Benchmark suite [KHCL07]: comparison of time in seconds (left) and number of iterations (right) used by `WP` and `NewST`.

Overall results

We limited the execution with 1 hour or 200 iterations of CEGAR per test case. Under this threshold 377 test cases completed by at least one of the techniques. In fact, 40% of them were completed in less than 2 seconds by all techniques and not more than 5 iterations. For these test cases the performance difference was not relevant and we exclude them from the comparison charts (if the opposite is not stated explicitly). For the remaining test cases SatAbs needs on average 42 predicates to perform a check, with a maximum of 177 predicates.

Only `NewST` was able to complete all of 377 considered test cases. `WP` did 9 less, while `SATQE` and `NewSP` failed to finish within a given limit on 76 and 26 test cases respectively. The complete results per individual benchmark are placed in Table A.1; the discussion in the next subsections is based on the aggregated scatter plots.

`WP` vs. `NewST`

The notable comparison of the two most effective methods — `WP` and `NewST` — gives a better understanding of the advantage of the new techniques. Figure 3.3 reports the scatter plots of the comparison. The results show that `NewST` almost systematically outperforms `WP`. In 98% of the test cases it requires fewer iterations to verify the property. Smaller number of iterations leads to reduction of the total verification time for 53% of the tests. On average, it decreased the total time by 42%, reaching more than double performance gain for some cases. For the small test cases (i.e., 5-10 iterations to complete) the application of the new technique does not give any significant advantage, but

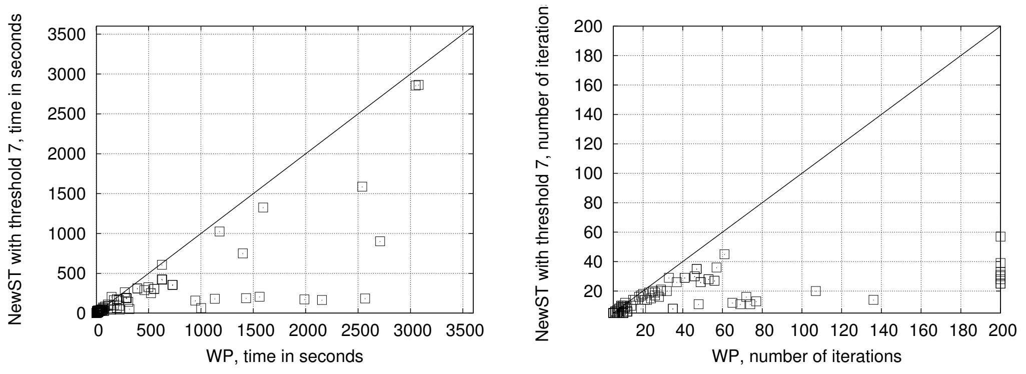


Figure 3.4. Benchmark suite [KHCL07]: scatter plot of time (left) and number of iterations (right) used by `WP` and `NewST` with a threshold.

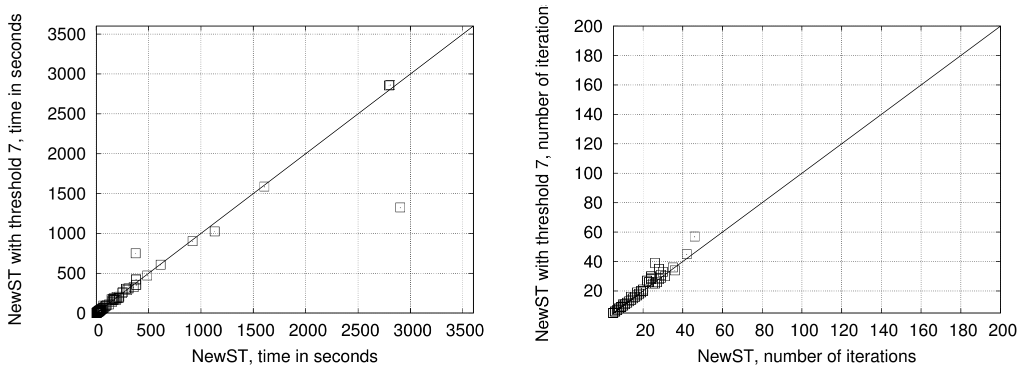


Figure 3.5. Benchmark suite [KHCL07]: comparison of time (left) and number of iterations (right) used by `NewST` and `NewST` with a threshold.

it becomes more pronounced with the growth of the test case complexity. The more time the model checking step in CEGAR requires, the bigger reduction in total time the CEGAR loop obtains due to fewer iterations.

3.4.3 A threshold for precise abstraction

In 47% of the test cases, where `NewST` was not better than `WP`, the difference in verification time usually was not bigger than 15%. As an exception, we found only one test case, in which the advantage of the smaller number of iterations was not able to compensate for the additional time spent for refinement (the point above the diagonal line in Figure 3.3, left).

We investigated the test case: for several program locations, the `Precise-`

`Abstraction` computation took longer than the time saved from the reduction in refinement iterations. This was due to the fact that the SAT-based enumeration of all spurious transitions was exponential in the number of predicates returned by `SpuriousTransition` (or `SpuriousPath`). Although there were only few transitions where it became critical, we decided to implement a heuristic, which would limit the application of precise computation. The heuristic forbids the application of `PreciseAbstraction` when the number of predicates reaches a given threshold N_σ. In such cases, `FastAbstraction` is applied instead of `PreciseAbstraction`. The value of the threshold depends on the application and the effectiveness of the predicate discovery techniques as well as the implementation of `PreciseAbstraction` and `FastAbstraction`.

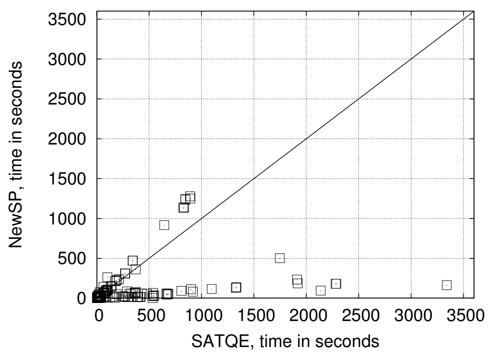
The idea can be further modified to use the already known threshold values. Separate limits can be set for `PreciseAbstraction` in the `SpuriousTransition` and `SpuriousPath` branches. In our experiments we used the pre-computed thresholds that seem optimal for the current implementation of the procedure: we use $N_{\sigma_{ST}} = 13$ for the call of `PreciseAbstraction` dedicated to the removal of spurious transition, while $N_{\sigma_{SP}} = 17$ when `PreciseAbstraction` is used to rule out spurious paths.

We can further optimize this approach by computing the threshold on-the-fly by limiting the maximum execution time for `PreciseAbstraction`: when the time-out is reached, the number of predicates that made the procedure blow up is used as a new threshold. The approach is shown in Algorithm 3.

We evaluated `NewST` with the pre-computed thresholds on the test suite from Ku et al. and obtained even better results than for pure `NewST`. The comparison of the `NewST` with the threshold against `WP` (Figure 3.4) shows that the improvement is systematic. The comparison between `NewST` with and without the threshold is shown in Figure 3.5. As expected, the results of both techniques are similar in more than 90% of the test cases, because the threshold was never reached and `FastAbstraction` was never applied. When the threshold was reached, the results of `NewST` with N_σ remained very close to the original `NewST`. But whenever the precise abstraction computation was a bottleneck, the use of the threshold enabled the use of the cheaper fast abstraction consequently resulting in a smaller computation time. The point below the diagonal line in Figure 3.5 (left) corresponds to one of the test cases where it happened. As an overall result `NewST` with a threshold reduced the total verification time by 5% compared to pure `NewST`.

Algorithm 3: The algorithm with localized abstraction and on-the-fly threshold computation. N_{TO} — time-out value for the `PreciseAbstraction`; N_σ — computed threshold value; $TimeoutWasReached$ — flag, which tracks if `PreciseAbstraction` was stopped by time-out N_σ.

```
MixCegarLoop(TransitionSystem M, Property F, Time $N_{TO}$)
begin
    $N_\sigma = unknown$;
    foreach $T$ in $M$ do $\Pi(T) = $ InitialPredicates($F$,$T$);
    foreach $T$ in $M$ do $\alpha(T) = $ FastAbstraction($T$,$\Pi$);
    while not TIMEOUT do
        $\pi = $ ModelCheck($\alpha$,$F$);
        if $\pi = \emptyset$ then return CORRECT;
        else
            $\sigma_{ST} = $ SpuriousTransition($\pi$);
            if $\sigma_{ST} \neq \emptyset$ then foreach $t \in \pi$ do
                $T = \tau(t)$;
                if $N_\sigma = unknown$ or $size(\sigma_{ST}(t)) < N_\sigma$ then
                    $C = $ PreciseAbstraction($T$,$\sigma_{ST}(t)$, $N_{TO}$);
                    if TimeoutWasReached then
                        $C = $ FastAbstraction($T$,$\sigma_{ST}(t)$);
                        $N_\sigma = size(\sigma_{ST}(t))$;
                    end if
                else $C = $ FastAbstraction($T$,$\sigma_{ST}(t)$);
                $\alpha(T) = \alpha(T) \wedge C$;
            end foreach
            else
                $\sigma_{SP} = $ SpuriousPath($\pi$);
                if $\sigma_{SP} = \emptyset$ then return INCORRECT;
                else foreach $t \in \pi$ do
                    $T = \tau(t)$;
                    $\Pi(T) = \Pi(T) \cup \sigma_{SP}(t)$;
                    if $N_\sigma = unknown$ or $size(\sigma_{SP}(t)) < N_\sigma$ then
                        $C = $ PreciseAbstraction($T$,$\sigma_{SP}(t)$, $N_{TO}$);
                        if TimeoutWasReached then
                            $C = $ FastAbstraction($T$,$\sigma_{SP}(t)$);
                            $N_\sigma = size(\sigma_{SP}(t))$;
                        end if
                    else $C = $ FastAbstraction($T$,$\sigma_{SP}(t)$);
                    $\alpha(T) = \alpha(T) \wedge C$;
                end foreach
            end if
        end if
    end while
end
```



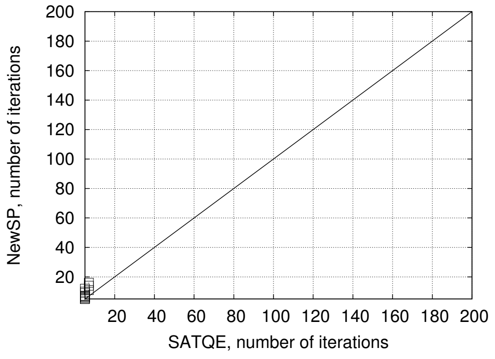
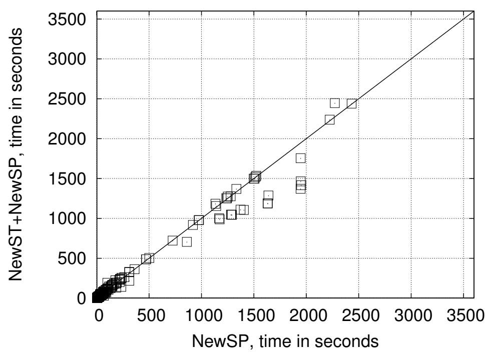


Figure 3.6. Benchmark suite [KHCL07]: scatter plot of time (left) and number of iterations (right) used by SATQE and NewSP.



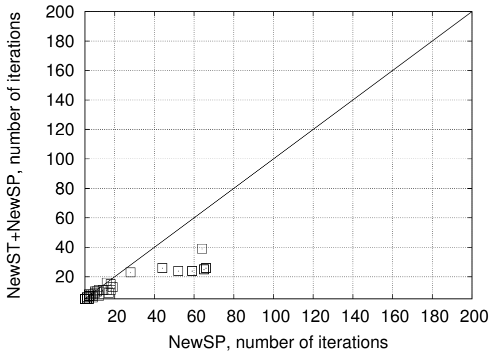
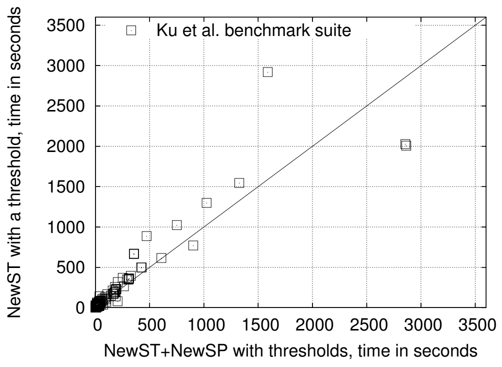


Figure 3.7. Benchmark suite [KHCL07]: scatter plot of time (left) and number of iterations (right) used by NewSP and NewST + NewSP.

SATQE, NewSP **and** NewST + NewSP

As expected, SATQE did not perform efficiently whenever a large number of predicates was involved in the abstraction. Although on smaller instances (≤ 30 predicates on average) it showed good results, on large instance it tended to time-out. Thus, it completed 76 fewer test cases than NewST. NewSP performed better (only 26 test cases were not finished) but still was worse than WP and NewST. The cause of the problem was similar to the one of SATQE or of NewST without a threshold: NewSP obtained too many predicates from SpuriousPath and the precise computation became very expensive. Nevertheless it scaled better than SATQE — see Figure 3.6 for comparison. Notice, that both techniques required fewer iterations than NewST and WP (Figure 3.3).



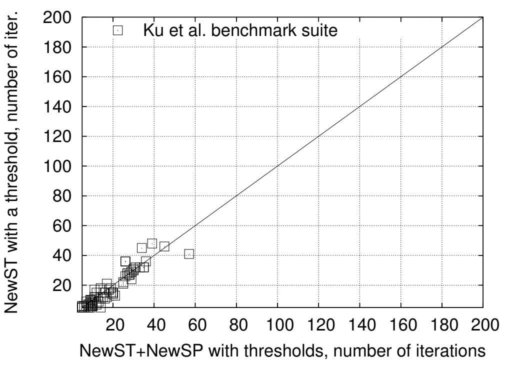


Figure 3.8. Benchmark suite [KHCL07]: scatter plot of time (left) and number of iterations (right) used by NewST and NewST + NewSP with thresholds.

The combination of NewST and NewSP outperformed NewSP (Figure 3.7). But the usage of PreciseAbstraction also caused the problem here and did not allow it to compete against NewST. Therefore a threshold for NewSP was also applied similar to its use in the NewST branch (Algorithm 3, lines 28–32).

We compared the fastest technique so far, NewST with a threshold, and a combination of NewSP and NewST with thresholds (Figure 3.8). However, on our test suite the winner was not obvious. Although the NewST + NewSP variant got more information from counterexamples to remove the spurious behaviors with (likely) cheap computation, the advantage over NewST was not enough to compensate for the additional call to precise abstraction computation. Nevertheless it confirmed that the use of a threshold helped to avoid problems caused by PreciseAbstraction.

3.4.4 Evaluation on industrial code

We experimented with the various programs from the open-source software packages like INN, WU-FTPD, GnuPG and others[5]. We applied the most effective methods — WP, NewST and NewST + NewSP with thresholds — and analyzed the programs for memory bounds violations.

The overall results on average repeated those from the benchmark suite with an exception that real programs had fewer trivial assertions. Here we report the outcome for one of the experiments. We analyzed the encode program from the inn utilities suite version 2.4.3 [inn]. It produces a seven-bit printable encoding of stdin on stdout and serves as a good example of a small

[5]All the benchmarks were taken from http://www.cprover.org/goto-cc/

memory-operating piece of C code. This program was taken as an example also because it is not very big (1.1KLOC) and has only 28 locations where a safety of the memory access should be checked. The size of the program allowed most of the claims to be verified within a one-hour time limit.

The results are reported in Table 3.2. For each claim and each technique we show total verification time and a number of the required refinement iterations. As expected reduction in the refinement iterations resulted in reduction of the total verification time. `NewST` used fewer refinements than `WP` in 12 out of 28 claims and won in verification time as well. Interesting to notice, the advantage was achieved any time more than 10 refinement iterations were required. For the other 16 claims, two techniques showed approximately the same result. Precise abstraction computation was localized and never required significant time. `NewST` + `NewSP` required fewer refinements than `WP` in all 28 claims and, as a result, it outperformed `WP` on all but 3 claims. However it did not perform better than `NewST` on every claim and therefore they are comparable in their advantages.

3.5 Related work and summary

This Chapter addresses the problem of refining the abstraction in the presence of spurious transitions. The solution was first given by Das and Dill [DD01] whose technique consists of removing one spurious transition at every refinement iteration. The approach may be very expensive because it requires a high number of iterations of the abstraction-refinement loop. In practice, the original technique of [DD01] is not feasible for real systems, but can be improved in a number of ways that will remove more spurious transitions at a time. For instance, Ball et al. [BCDR04] improved the refinement by strengthening the condition added to the transition relation to remove more spurious transitions. Their idea is to syntactically simplify the condition and to check if a larger set of spurious transitions is found. The technique cannot guarantee the removal of all spurious transitions relevant to a detected counterexample as achieved by our method. But, due to the syntactic nature of the analysis, it is computationally not very expensive and, thus, can be included as a fast refinement in our combined algorithm.

In [CTVW04, JKSC05, JIGG05], a different technique is presented based on the use of SAT techniques. Transitions are simulated over the concrete program by means of SAT formulæ. If the transition is not concretizable the SAT solver will produce a resolution proof of unsatisfiability. It is then possible to extract

	Total time, in seconds			Number of iterations		
Claim #	`WP`	`NewST`	`NewST` + `NewSP`	`WP`	`NewST`	`NewST` + `NewSP`
1	3.478	3.464	2.871	5	5	4
2	2.243	2.318	1.892	4	4	3
3	7.977	8.345	6.640	6	6	5
4	124.013	104.657	83.893	25	19	10
5	4.149	4.222	3.529	4	4	3
6	137.317	97.449	121.919	28	17	12
7	2.683	2.698	1.567	3	3	2
8	2.712	2.636	1.594	3	3	2
9	37.860	28.783	31.429	10	8	7
10	27.575	27.225	29.612	9	9	7
11	5.975	5.801	4.727	6	6	4
12	76.945	49.822	71.106	13	10	10
13	TO	TO	TO	TO	TO	TO
14	7.894	8.195	6.985	6	6	5
15	128.271	98.010	88.266	26	19	10
16	4.207	4.261	3.420	4	4	3
17	145.884	112.898	122.006	30	19	13
18	2.113	2.123	1.330	3	3	2
19	2.193	2.158	1.370	3	3	2
20	31.598	22.788	27.131	9	7	6
21	27.163	22.906	28.050	10	8	6
22	4.349	4.495	3.111	5	5	3
23	77.919	49.293	67.942	13	10	10
24	10.981	9.494	11.103	8	7	6
25	7.408	7.603	6.620	6	6	5
26	0.151	0.124	0.113	32	23	14
27	4.439	4.393	3.592	4	4	3
28	125.827	73.210	97.236	30	15	14

Table 3.2. `inn-encode 2.4.3` program: total time and number of refinement iterations required to verify 28 memory-bounds claims (automatically planted by SATABS); TO stands for time-out (3600 sec.).

from the proof either a core set of predicates or a constraint sufficient to remove the spurious transition. Though, in principle, the technique can remove many spurious transitions at once, the efficiency strongly depends on the unsatisfiability proof. In the worst case, it may require a number of abstraction refinements exponential in the number of predicates.

The technique of [JM05] also exploits the unsatisfiability proof but it is based on interpolation. The interpolant produced by the proof is indeed an over-approximation of the exact abstraction able to remove the spurious transition. As in the case of unsat cores, the technique depends on the heuristics to produce unsatisfiability proofs. The interpolant is not always strong enough to remove all spurious transitions as done in our algorithm.

This work instead proposes a greedy approach where all spurious transitions between two locations are removed. The idea is that the computation can be efficient because it is localized and on-demand. The technique inherits the efficiency of the approximated abstraction which is used any time new predicates are discovered. At the same time, the precision of the minimal abstraction is exploited whenever spurious transitions are found.

In future it would be interesting to implement the combined fast/precise abstraction approach in tools that are based on interpolation for predicate discovery [HJMM04, JM06] and investigate the same trade-off between precise and approximated approaches in the context of purely interpolation-based model checking[McM06], which does not need predicate abstraction.

Summary

This chapter presented a new approach to the abstraction refinement that combines precise and approximated techniques. First, the proposed algorithm benefits from the precise component, because it avoids too many iterations due to spurious transitions of the abstract model. Second, it uses the fast component to discover the spurious counterexample. Moreover, by exploiting the localized-abstraction framework, it reduces the abstraction computation to the parts of the system that are relevant to the property and keeps the approximated abstraction in all parts of the program that are irrelevant to prove the property. Our technique is independent of any particular abstraction or refinement procedure and can be used for any combination of the existing abstraction and refinement techniques.

The algorithm was implemented in SatAbs software model checker and is available for experiments by other researchers.

We performed an extensive evaluation on programs of various sizes comparing the new technique with the classical precise and imprecise algorithms. Our tests with various benchmarks show that the new approach systematically outperforms both precise and imprecise techniques. Altogether, it confirms that our new technique achieves the goal of reducing the number of iterations of the CEGAR loop.

We notice that our techniques are particularly efficient when the approximated abstraction produces a large number of spurious transitions and, at the same time, hand the precise abstraction does not blow up. It is clear that our techniques always take a number of iterations that lies between the ones used by the precise abstraction and the ones used by the approximated abstraction. The difference of iterations between the fast and the precise abstraction is due to the spurious transition refinement step. As expected, our technique removed the necessary spurious transitions in fewer iterations.

Chapter 4

Program Summarization using Abstract Transformers

Understanding is but the sum of
our misunderstandings.
Haruki Murakami

This chapter tackles the problem of scalable and precise program abstraction. We identify loops in a program as the main obstacle for the successful application of the existing abstraction methods, especially those that rely on the fixpoint-based over-approximation computation.

To address this problem we present an algorithm that constructs an over-approximation of the program's set of reachable states by replacing loops in the control-flow graph with their abstract transformer. For each loop, starting from the inner-most one, the abstract transformer is obtained by 1) employing a (problem-specific) abstract domain to generate candidate assertions and 2) checking if such an assertion is a loop invariant. The algorithm run-time is linear in the number of loops and relies on a finite number of relatively cheap calls to first-order decision procedures. It also allows localizing the application of abstract domains to an individual loop, thus helping to improve the precision of the constructed abstraction.

4.1 Introduction

As described before in Section 2.5.2, *abstract interpretation* [CC77] is one of the approaches that enables practical program verification. It is a commonly-used framework for the approximative analysis of discrete transition systems, and it is based on fixpoint computations. In abstract interpretation, the behavior of a program is evaluated over the abstract domain using an *abstract transformer* — predicate transformer that reflects the semantics of program instructions into mutation of an abstract value. This is iterated until the set of abstract states saturates, i.e., an abstract fixpoint is reached. This abstract fixpoint is guaranteed to be an over-approximation of the set of reachable states of the original program with regards to employed abstract domains.

A main issue in employing abstract interpretation is the number of iterations required to reach the abstract fixpoint. On large benchmarks, thousands of iterations are commonly observed, even when using simplistic abstract domains. Thus, many tools implementing abstract interpretation apply *widening* in order to accelerate convergence. Widening, however, may yield imprecision, and thus, the abstract fixpoint may not be strong enough to prove the desired property [CC92].

This Chapter presents a novel technique to address this problem, which uses a *symbolic abstract transformer* [RSY04]. A symbolic abstract transformer for a given program fragment is a relation over a pair of abstract states $\hat{s}, \hat{s}'$ that holds if the fragment transforms $\hat{s}$ into $\hat{s}'$. We propose to apply the transformer to perform sound *summarization*, i.e., to replace parts of the program by a smaller representative. In particular, we use the transformer to summarize loops and (recursion-free) function calls.

The symbolic abstract transformer is usually computed by checking if a given abstract transition is consistent with the semantics of a program statement [RSY04, CKSY04]. Our technique generalizes the abstract transformer computation and applies it to program fragments: given an abstract transition relation, we check if it is consistent with the program semantics. This way, we can tailor the abstraction to each program fragment. In particular, for loop-free programs, we precisely encode their semantics into symbolic formulas. For loops, we exploit the symbolic transformer of the loop body to infer invariants of the loop. This is implemented by means of a sequence of calls to a decision procedure (i.e., SAT- or SMT-solver) for the chosen program logic.

When applied starting from the inner-most loops and the leaves of the call graph, the run-time of the resulting procedure becomes linear in the number of looping constructs in the program, and thus, is often much smaller than the number of iterations required by the traditional saturation procedure. We show soundness of the procedure and discuss its precision compared to the conventional approach on a given abstract domain. In case the property fails, a diagnostic counterexample can be obtained, which we call *leaping counterexample*. This diagnostic information is often very helpful for understanding the nature of the problem, and is considered a major plus for program analysis tools. Additionally, our technique *localizes* the abstract domains: we use different abstract domains for different parts of the code. This further improves the scalability of the analysis.

We implemented the loop summarization procedure in a tool called LOOPFROG and applied it to search for buffer-overflow errors in well-known UNIX programs. Our experimental results demonstrate that the procedure is more precise than other static analysis tools tailored to analysis of buffer overruns. Moreover, it scales gracefully to large programs even if complex abstract domains are used.

This Chapter is structured as follows. Section 4.2 formalizes the notion of abstract interpretation and abstract transformer for a program. Section 4.3 first describes construction of an abstract transformer for loop-free and single-loop fragments of the program. Next, it generalizes the procedure to an arbitrary program and presents a loop summarization algorithm. Section 4.4 details the LOOPFROG implementation. Section 4.5 demonstrates the experimental evaluation and, finally, Section 4.6 discuses the related work and summarizes the results.

4.2 Abstract interpretation

We use program formalization as described in Section 2.1, i.e., a program P is defined using a pair $\langle U, G\rangle$, where U is the universe, from which the values of the program variables are drawn, and G is a program graph.

The (concrete) semantics of a program is given by the pair $\langle A, \tau\rangle$, where:

- A is the set of *assertions* of the program, where each assertion $p \in A$ is a predicate over U; $A(\Rightarrow, false, true, \vee, \wedge)$ is a complete Boolean lattice;
- $\tau : L \rightarrow (A \rightarrow A)$ is the predicate transformer.

An *abstract interpretation* is a pair $\langle \hat{A}, t\rangle$, where $\hat{A}$ is a complete lattice $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, and $t : L \rightarrow (\hat{A} \rightarrow \hat{A})$ is a predicate transformer [CC79]. Note that $\langle A, \tau\rangle$ is a particular abstract interpretation called the *concrete interpretation*. In the following, we assume that for every command $c \in L$, the function $t(c)$ is monotone (which is the case for all natural predicate transformers). Given a predicate transformer t, the function $\tilde{t} : L^* \rightarrow (\hat{A} \rightarrow \hat{A})$ is recursively defined as follows:

$$\tilde{t}(p)(\phi) = \begin{cases} \phi & \text{if } p \text{ is empty} \\ \tilde{t}(e)(t(q)(\phi)) & \text{if } p = q; e \text{ for some } q \in L,\ e \in L^*. \end{cases}$$

Example 4.1. *We continue using the program in Figure 2.1. Consider an abstract domain where abstract states are a four-tuple $\langle p_a, z_a, s_a, l_a \rangle$. The first member, p_a is the offset of the pointer p from the base address of the array a (i.e., $p - a$ in our example), the Boolean z_a holds if a contains the zero character, the Boolean s_a holds if a contains the slash character, l_a is the index of the first zero character if present. The predicate transformer t is defined as follows:*

$$\begin{array}{ll}
t(p = a)(\phi) = \phi[p_a := 0] & \textit{for any assertion } \phi; \\
t(*p\,!= 0)(\phi) = \phi \wedge (p_a \neq l_a) & \textit{for any assertion } \phi; \\
t(*p == 0)(\phi) = \phi \wedge z_a \wedge (p_a \geq l_a) & \textit{for any assertion } \phi; \\
t(*p ==' /')(\phi) = \phi \wedge s_a & \textit{for any assertion } \phi; \\
t(*p\,!=' /')(\phi) = \phi & \textit{for any assertion } \phi; \\
t(*p = 0)(\phi) = \begin{cases} \phi[z_a := true, l_a := p_a] & \textit{if } \phi \Rightarrow (p_a < l_a) \\ \phi[z_a := true] & \textit{otherwise}; \end{cases} & \\
t(p{+}{+})(\phi) = \phi[p_a := p_a + 1] & \textit{for any assertion } \phi.
\end{array}$$

(We used $\phi[x := v]$ to denote an assertion equal to ϕ apart from the variable x that takes value v.)

Given a program P, an abstract interpretation $\langle \hat{A}, t \rangle$, and an element $\phi \in \hat{A}$, we define the *Merge Over all Paths $MOP_P(t, \phi)$* as

$$MOP_P(t, \phi) := \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\phi)\,.$$

Given two complete lattices $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $\hat{A}'(\sqsubseteq', \bot', \top', \sqcup', \sqcap')$, the pair of functions $\langle \alpha, \gamma \rangle$, with $\alpha : \hat{A} \to \hat{A}'$ and $\gamma : \hat{A}' \to \hat{A}$ is a *Galois connection* iff α and γ are monotone and satisfy:

$$\begin{array}{lll}
\text{for all } \phi \in \hat{A}: & \phi & \sqsubseteq \gamma(\alpha(\phi)) \\
\text{for all } \phi' \in \hat{A}': & \alpha(\gamma(\phi')) & \sqsubseteq' \phi'\,.
\end{array}$$

An abstract interpretation $\langle \hat{A}, t \rangle$ is a *correct over-approximation* of the concrete interpretation $\langle A, \tau \rangle$ iff there exists a Galois connection $\langle \alpha, \gamma \rangle$ such that for all $\phi \in \hat{A}$ and $p \in A$, if $p \Rightarrow \gamma(\phi)$, then $\alpha(MOP_P(\tau, p)) \sqsubseteq MOP_P(t, \phi)$ (i.e., $MOP_P(\tau, p) \Rightarrow \gamma(MOP_P(t, \phi))$).

4.2.1 Approaches to computation of an abstract transformer

In order to implement abstract interpretation for a given abstract domain, an algorithmic description of the abstract predicate transformer $t(p)$ for a specific command $p \in L$ is required. Reps et al. describe an algorithm that implements the *best possible* (i.e., most precise) abstract transformer for a given finite-height abstract domain [RSY04]. Graf and Saïdi's algorithm for constructing predicate abstractions [GS97] is identified as a special case.

The algorithm has two inputs: a formula $F_{\tau(q)}$, which represents a command $q \in L$ symbolically, and an assertion $\phi \in \hat{A}$. It returns the image of the predicate transformer $t(q)(\phi)$. The formula $F_{\tau(q)}$ is passed to a decision procedure, which is expected to provide a satisfying assignment to the variables. The assignment represents one concrete transition $p, p' \in A$. The transition is abstracted into a pair $\phi, \phi' \in \hat{A}$, and a blocking constraint is added to remove this satisfying assignment. The algorithm iterates until the formula becomes unsatisfiable. An instance of the algorithm for the case of predicate abstraction is the implementation of SatAbs described in [CKSY04]. SatAbs uses a propositional SAT-solver as decision procedure for bit-vector arithmetic. The procedure is worst-case exponential in the number of predicates, and thus, alternatives have been explored. In [LBC05, KS06] a symbolic decision procedure generates a symbolic formula that represents the set of all solutions. In [LNO06], a first-order formula is used and the computation of all solutions is carried out by an SMT-solver. In [CCF$^+$07], a similar technique is proposed where BDDs are used in order to efficiently deal with the Boolean component of $F_{\tau(q)}$.

4.3 Summarization using symbolic abstract transformers

In the following four subsections, we describe the steps of our summarization approach. We first define summarization as an over-approximation of a code fragment. Next, we show that a precise summary can be computed for a loop-free code fragment, and we explain how a precise summary of a loop body is used to obtain information about the loop. Finally, we give a bottom-up summarization algorithm for arbitrary programs.

4.3.1 Abstract summarization

The idea of summarization is to replace a code fragment, e.g., a procedure of the program, by a *summary*, which is a representation of the fragment. Computing an exact summary of a program (fragment) is in general undecidable. We therefore settle for an over-approximation. We formalize the conditions the summary must fulfill in order to have a semantics that over-approximates the original program.

We extend the definition of a correct over-approximation (from Sec. 4.2) to programs. Given two programs P and P' on the same universe U, we say that P' is a *correct over-approximation* of P iff for all $p \in A(\Rightarrow, false, true, \vee, \wedge)$, $MOP_P(\tau, p) \Rightarrow MOP_{P'}(\tau, p)$.

Definition 15. *[Abstract Summary] Given a program P, and an abstract interpretation $\langle \hat{A}, t\rangle$ with a Galois connection $\langle \alpha, \gamma\rangle$ with $\langle A, \tau\rangle$, we denote the* abstract summary *of P by $Sum_{\langle \hat{A},t\rangle}(P)$. It is defined as the program $\langle U, G\rangle$, where $G = \langle \{v_i, v_o\}, \{\langle v_i, v_o\rangle\}, v_i, v_o, \{a\}, C\rangle$ and $\{a\}$ together with $C(\langle v_i, v_o\rangle) \rightarrow \{a\}$ is a new (concrete) command a such that $\tau(a)(p) = \gamma(MOP_P(t, \alpha(p)))$.*

Lemma 1. *If $\langle \hat{A}, t\rangle$ is a correct over-approximation of $\langle A, \tau\rangle$, the abstract summary $Sum_{\langle \hat{A},t\rangle}(P)$ is a correct over-approximation of P.*

Proof. Let $P' = Sum_{\langle \hat{A},t\rangle}(P)$. For all $p \in A$,

$$\begin{array}{rcll} MOP_P(\tau, p)) & \Rightarrow & \gamma(MOP_P(t, \alpha(p))) & \text{[by definition of correct over-approximation,} \\ & & & \text{page 56, Section 4.2, last paragraph]} \\ & = & \tau(a)(p) & \text{[by Definition 15]} \\ & = & MOP_{P'}(\tau, p) & \text{[MOP over a single-command path } a \text{ in } P'\text{].} \end{array}$$

□

The next sections discuss our algorithms for computing abstract summaries. Summarization technique is first applied to particular fragments of the program, specifically to loop-free (Section 4.3.2) and single-loop programs (Section 4.3.3). In Section 4.3.4, we use these procedures as subroutines to obtain the summarization of an arbitrary program. We formalize code fragments as *program sub-graphs*.

Definition 16. *Given two program graphs $G = \langle V, E, v_i, v_o, L, C\rangle$ and $G' = \langle V', E', v_i', v_o', L', C'\rangle$, G' is a* program sub-graph *of G iff $V' \subseteq V$, $E' \subseteq E$, and $C'(e) = C(e)$ for every edge $e \in E'$.*

4.3.2 Summarization of loop-free programs

Obtaining $MOP_P(t, \phi)$ is as hard as assertion checking on the original program. Nevertheless, there are restricted cases where it is possible to represent $MOP_P(t, \phi)$ using a symbolic predicate transformer.

Let us consider a program P with a finite number of paths, in particular, a program whose program graph does not contain any cycle. A program graph $G = \langle V, E, v_i, v_o, L, C\rangle$ is *loop free* iff G is a directed acyclic graph.

In the case of a loop-free program P, we can compute a precise (not abstract) summary by means of a formula F_P that represents the concrete behavior of P. This formula is obtained by converting P to a static single assignment (SSA) form, whose size is at most quadratic in the size of P (the details of this step are beyond the scope of this work; see [CKL04] for details).

Example 4.2. *We continue the running example (Fig. 2.1). The symbolic transformer of the loop body P' is represented by:*
$((*p =' /' \wedge a' = a[*p = 0]) \vee (*p \neq' /' \wedge a' = a)) \wedge (p' = p + 1)$.

Recall the abstract domain from Ex. 4.1. We can deduce that:

1. *if $m < n$, then $MOP_{P'}(t, (p_a = m \wedge z_a \wedge (l_a = n) \wedge \neg s_a)) = (p_a = m + 1 \wedge z_a \wedge l_a = n \wedge \neg s_a)$*

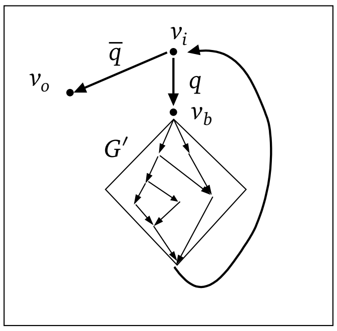
2. *$MOP_{P'}(t, z_a) = z_a$.*

This example highlights the generic nature of our technique. For instance, case 1 of the example cannot be obtained by means of predicate abstraction because it requires an infinite number of predicates. Also, the algorithm presented in [RSY04] cannot handle this example because assuming the string length has no a-priori bound, the lattice of the abstract interpretation has infinite height.

4.3.3 Summarization of single-loop programs

We now consider a program that consists of a single loop.

Definition 17. *A program $P = \langle U, G\rangle$ is a* single-loop program *iff $G = \langle V, E, v_i, v_o, L, C\rangle$ and there exists a program sub-graph G' and a test $q \in L_T$ such that*



- *$G' = \langle V', E', v_b, v_i, L', C' \rangle$ with*
  - *$V' = V \setminus \{v_o\}$,*
  - *$E' = E \setminus \{\langle v_i, v_o \rangle, \langle v_i, v_b \rangle\}$,*
  - *$L' = L \setminus q$,*
  - *$C'(e) = C(e)$ for all $e \in E'$,*
  - *G' is loop free.*
- *$C(\langle v_i, v_b \rangle) = q$, $C(\langle v_i, v_o \rangle) = \overline{q}$.*

The following can be seen as the “abstract interpretation analog” of Hoare’s rule for `while` loops.

Theorem 4. *Given a single-loop program P with guard q and loop body P', and an abstract interpretation $\langle \hat{A}, t \rangle$, let ψ be an assertion satisfying $MOP_{P'}(t, t(q)(\psi)) \sqsubseteq \psi$ and let $\langle \hat{A}, t_\psi \rangle$ be a new abstract interpretation s.t.*

$$MOP_P(t_\psi, \phi) = \begin{cases} t(\overline{q})(\psi) & \text{if } \phi \sqsubseteq \psi \\ \top & \text{elsewhere.} \end{cases}$$

If $\langle \hat{A}, t \rangle$ is a correct over-approximation, then $\langle \hat{A}, t_\psi \rangle$ is a correct over-approximation as well.

We first record the following simple lemma.

Lemma 2. *Given a loop-free program P, and an abstract interpretation $\langle \hat{A}, t \rangle$, if $MOP_P(t, \psi) \sqsubseteq \psi$, then, for all repetitions of loop-free paths of a program P, i.e., for all $\pi \in (paths(P))^*$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$.*

Proof. If $MOP_P(t, \psi) = \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\psi) \sqsubseteq \psi$, then, for all paths $\pi \in paths(P)$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$. Thus, by induction on repetitions of loop-free paths, for all paths $\pi \in (paths(P))^*$, $\tilde{t}(\pi)(\psi) \sqsubseteq \psi$. □

Proof of Theorem 4. If $\phi \sqsubseteq \psi$,

$$\begin{array}{rcll}
\alpha(MOP_P(\tau,p)) & \sqsubseteq & MOP_P(t,p) & [\langle \hat{A}, t\rangle \text{ is a correct over-approximation}] \\
& = & \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\phi) & [\text{by definition of } MOP] \\
& \sqsubseteq & \bigsqcup_{\pi \in paths(P)} \tilde{t}(\pi)(\psi) & [\phi \sqsubseteq \psi] \\
& = & \bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\pi';\overline{q})^*)(\psi) & [P \text{ is a single-loop program}] \\
& = & \bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\overline{q})^*)(\tilde{t}((\pi')^*)(\psi)) & [\text{by definition of } \tilde{t}] \\
& \sqsubseteq & t(\overline{q})(\bigsqcup_{\pi \in (q;\pi')^*, \pi' \in paths(P')} \tilde{t}((\pi')^*)(\psi)) & [t \text{ is monotone}] \\
& \sqsubseteq & t(\overline{q})(\psi) & [\text{by Lemma 2}]
\end{array}$$

Otherwise, trivially $\alpha(MOP_P(\tau,p)) \sqsubseteq \top = MOP_P(t_\psi, \phi)$. □

In other words, if we apply the predicate transformer of the test q and then the transformer of the loop body P' to the assertion ψ, and we obtain an assertion at least as strong as ψ, then ψ is an invariant of the loop. If a stronger assertion ϕ holds before the loop, the predicate transformer of $\overline{q}$ applied to ϕ holds afterwards.

Theorem 4 gives rise to a summarization algorithm. Given a program fragment and an abstract domain, we heuristically provide a set of formulas that encode that a (possibly infinite) set of assertions ψ are invariant (for example, $x' = x$ encodes that every ψ defined as $x = c$, with c a value in the domain U, is an invariant); we apply a decision procedure to check if the formulas are satisfiable.

The construction of the summary is then straightforward: given a single-loop program P, an abstract interpretation $\langle \hat{A}, t\rangle$, and an invariant ψ for the loop body, let $\langle \hat{A}, t_\psi\rangle$ be the abstract interpretation as defined in Theorem 4. We denote the summary $Sum_{\langle \hat{A}, t_\psi\rangle}(P)$ by $\texttt{SlS}(P, \hat{A}, t_\psi)$ (Single-Loop Summary).

Corollary 1. *If $\langle \hat{A}, t\rangle$ is a correct over-approximation of $\langle A, \tau\rangle$, then $\texttt{SlS}(P, \hat{A}, t_\psi)$ is a correct over-approximation of P.*

Example 4.3. *We continue the running example from Figure 2.1. Recall the abstract domain in Ex. 4.1. Let P' denote the loop body of the example program and let q denote the loop guard. By applying the symbolic transformer from Ex. 4.2, we can check that the following conditions hold:*

1. *$MOP_{P'}(t, t(q)(\phi)) \sqsubseteq \phi$ for any assertion $((p_a \leq l_a) \wedge z_a \wedge \neg s_a)$.*

2. *$MOP_{P'}(t, t(q)(\phi)) \sqsubseteq \phi$ for the assertion z_a.*

Thus, we summarize the loop with the following predicate transformer:

$$(z_a \rightarrow z'_a) \wedge (((p_a \leq l_a) \wedge z_a \wedge \neg s_a) \rightarrow ((p'_a = l'_a) \wedge z'_a \wedge \neg s'_a)) \; .$$

4.3.4 Summarization for arbitrary programs

We now describe an algorithm for over-approximating an arbitrary program. Like traditional algorithms (e.g., [Tar81]), the dependency tree of program fragments is traversed bottom-up, starting from the leaves. The code fragments we consider may be function calls or loops. We treat function calls as arbitrary sub-graphs (see Def. 16) of the program graph, and do not allow recursion. We support irreducible graphs using loop simulation [AM79].

Specifically, we define the *sub-graph dependency tree* of a program $P = \langle U, G\rangle$ as the tree $\langle T, >\rangle$, where:

- the set of nodes of the tree are program sub-graphs of G;
- for $G_1, G_2 \in T$, $G_1 > G_2$ iff G_2 is a program sub-graph of G_1 with $G_1 \neq G_2$;
- the root of the tree is G;
- every leaf is a loop-free or single-loop sub-graph;
- every loop sub-graph is in T.

Algorithm 4 takes a program as input and computes its summary by following the structure of the sub-graph dependency tree (Line 3). Thus, the algorithm is called recursively on the sub-program until a leaf is found (Line 5). If it is a single loop, an abstract domain is chosen (Line 11) and the loop is summarized as described in Section 4.3.3 (Line 13). If it is a loop-free program, it is summarized with a symbolic transformer as described in Section 4.3.2 (Line 15). Finally, the old sub-program is replaced with its summary (Line 7) and the sub-graph dependency tree is updated (Line 8). Eventually, the entire program is summarized[1].

[1]Algorithm 6 in Section 5.2.2 presents the arbitrary program summarization split into subroutines

Algorithm 4: Arbitrary program summarization

```
1  SUMMARIZE($P$)
   input  : program $P = \langle U, G\rangle$
   output: over-approximation $P'$ of $P$
2  begin
3      $\langle T, >\rangle$ :=sub-graph dependency tree of $P$;
4      $P_r := P$;
5      for each $G'$ such that $G > G'$ do
6          $\langle U, G''\rangle$:=SUMMARIZE($\langle U, G'\rangle$);
7          $P_r := P_r$ where $G'$ is replaced with $G''$;
8          update $\langle T, >\rangle$;
9      end for
10     if $P_r$ is a single loop then
11         $\langle \hat{A}, t\rangle$ := choose abstract interpretation for $P_r$;
           /* Choice of abstract interpretation defines set of
           candidate assertions $\psi$, which are checked to hold in
           the next step.                         */;
12         $\psi$ := test invariant candidates for $t$ on $P_r$;
13         $P' :=$ SlS($P_r$,$\hat{A}$, $t_\psi$);
           /* Those $\psi$ that hold on $P_r$ form the single-loop
           summary (SlS).                         */;
14     else
           /* $P_r$ is loop-free                     */;
15         $P' := Sum_{\langle A,\tau\rangle}(P_r)$;
16     end if
17     return $P'$
18 end
```

Theorem 5. SUMMARIZE(P) *is a correct over-approximation of P.*

Proof. We prove the theorem by induction on the structure of the sub-graph dependency tree.

In the first base case (P_r is loop-free) summary is precise by construction and, thus, is a correct over-approximation of P.

In the second base case (P_r is a single loop), by hypothesis, each abstract interpretation chosen at Line 11 is a correct over-approximation of the concrete interpretation. Thus, if P is a single-loop or a loop-free program, P' is a correct over-approximation of P (resp. by Theorem 4 and by definition of abstract summary).

In the inductive case, we select a program subgraph G' and we replace it with G'', where $\langle U, G''\rangle$=Summarize($\langle U, G'\rangle$). By inductive hypothesis, we know that $\langle U, G''\rangle$ is a correct over-approximation of $\langle U, G'\rangle$. Thus, for all $p \in A$, $MOP_{\langle U,G'\rangle}(\tau, p) \Rightarrow MOP_{\langle U,G''\rangle}(\tau, p)$. Note that G'' contains only a single command g.

We want to prove that for all $p \in A$, $MOP_P(\tau, p) \Rightarrow MOP_{P'}(\tau, p)$ (for readability, we replace the subscript "$(\pi_i; \pi_g; \pi_f) \in paths(P)$, $\pi_g \in paths(\langle U, G'\rangle)$, and $\pi_i \cap G' = \emptyset$" with $*$ and "$\pi \in paths(P)$, and $\pi \cap G' = \emptyset$" with $**$):

$$
\begin{aligned}
MOP_P(\tau, p) \quad = \quad & \bigsqcup_{\pi \in paths(P)} \tilde{\tau}(\pi)(p) \quad \text{[by definition of } MOP\text{]} \\
= & \bigsqcup_{*} \tilde{\tau}(\pi_f)(\tilde{\tau}(\pi_g)(\tilde{\tau}(\pi_i)(p))) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \quad [G' \text{ is a subgraph}] \\
\Rightarrow & \bigsqcup_{*} \tilde{\tau}(\pi_f)(MOP_{\langle U,G''\rangle}(\tilde{\tau}, (\tilde{\tau}(\pi_i)(p)))) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \; [G'' \text{ is an over-approx.}] \\
= & \bigsqcup_{*} \tilde{\tau}(\pi_f)(MOP_{\langle U,(G'';\pi_i)\rangle}(\tau, p)) \cup \bigsqcup_{**} \tilde{\tau}(\pi)(p) \quad \text{[by definition of } MOP\text{]} \\
= & \bigsqcup_{\pi \in paths(P)} MOP_{\pi[g/\pi_g]}(\tau, p) \quad \text{[by induction on length of paths]} \\
= & \bigsqcup_{\pi \in paths(P')} MOP_{\pi}(\tau, p) \quad \text{[by definition of } \pi'] \\
= & MOP_{\pi'}(\tau, P) \quad \text{[by definition of } MOP\text{]}
\end{aligned}
$$

□

The precision of the over-approximation is controlled by the precision of the symbolic transformers. However, in general, the computation of the best abstract transformer is an expensive iterative procedure. We use the inexpensive syntactic procedure for loop-free fragments. Loss of precision only happens when summarizing loops, and greatly depends on the abstract interpretation chosen in Line 11.

Note that Algorithm 4 does not limit the selection of abstract domains to any specific type of domains, and that it does not iterate the predicate transformer on the program. Furthermore, this algorithm allows for *localization* of the summarization procedure, as a new domain may be chosen for every loop. Once the domains are chosen, it is also easy to monitor the progress of the summarization, as the number of loops and the cost of computing the symbolic transformers are known – another distinguishing feature of our algorithm.

The summarization can serve as an over-approximation of the program. It can be trivially analyzed to prove unreachability, or equivalently, to prove assertions.

4.3.5 Leaping counterexamples

Let P' denote the summary of the program. The program P' is a loop-free sequence of symbolic summaries for loop-free fragments and loop summaries. A *counterexample* for an assertion in P' follows this structure: when traversing symbolic summaries for loop-free fragments, it is identical to a concrete counterexample. Upon entering a loop summary, the effect of the loop body is given as a single transition in the counterexample: we say that the counterexample *leaps* over the loop.

Example 4.4. *Consider the summary from Ex. 4.3. Suppose that in the initial condition, the buffer a contains a null terminating character in position n and no '/' character. If we check that, after the loop, p_a is greater than the size n, we obtain a counterexample with $p_a^0 = 0, p_a^1 = n$.*

The *leaping counterexample* may only exist with respect to the abstract interpretations used to summarize the loops, i.e., they may be spurious in the concrete interpretation. Nevertheless, they provide useful diagnostic feedback to the programmer, as they show a (partial) path to the violated assertion, and contain many of the input values the program needs to read to violate the assertion. Furthermore, spurious counterexamples can be eliminated by combining our technique with counterexample-guided abstraction refinement, as we do have an abstract counterexample.

4.4 Loopfrog

The theoretical concept of symbolic abstract transformers is implemented and put to use by our tool LOOPFROG. The architecture is outlined in Figure 4.1. As input, LOOPFROG receives a model file, extracted from software sources by GOTO-CC[2]. This model extractor features full ANSI-C support and simplifies verification of software projects that require complex build systems. It mimics the behavior of the compiler, and thus 'compiles' a model file using the original settings and options. Switching from compilation mode to verification mode is thus achieved by changing a single option in the build system. As suggested by Figure 4.1, all other steps are fully automated.

The resulting model contains a control flow graph and a symbol table, i.e., it is an intermediate representation of the original program in a single file. For calls to system library functions, abstractions containing assertions (pre-condition checks) and assumptions (post-conditions) are inserted. Note that the model also contains the properties to be checked in the form of assertions (calls to the ASSERT function).

Preprocessing

The instrumented model is what is passed to the first stage of LOOPFROG. In this preprocessing stage, the model is adjusted in various ways to increase performance and precision. First, irreducible control flow graphs are rewritten according to an algorithm due to Ashcroft and Manna [AM79]. Like in a compiler, small functions are inlined. This increases the model size, but also improves the precision of subsequent analysis. After this, LOOPFROG runs a field-sensitive pointer analysis. The information obtained through this is used to insert assertions over pointers, and to eliminate pointer variables in the program where possible. On request, LOOPFROG automatically adds assertions to verify the correctness of pointer operations, array bounds, and arithmetic overflows.

Loop summarization phase

Once the preprocessing is finished, LOOPFROG starts to replace loops in the program with summaries. These are shorter, loop-less program fragments that over-approximate the original program behavior. To accomplish this soundly, as described in Section 4.3, each loop is replaced with a loop-less piece of code that 'havocs' the program state, i.e., it resets all variables that may be

[2]`http://www.cprover.org/goto-cc/`

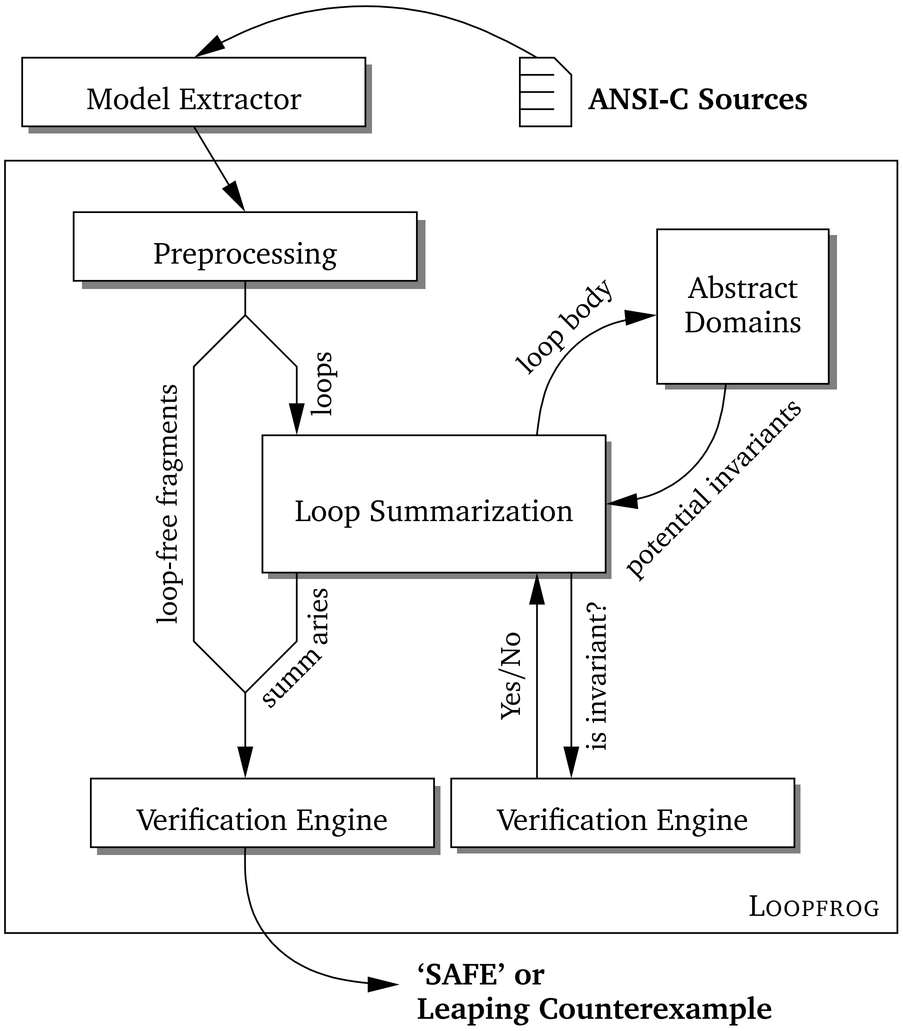


Figure 4.1. Architecture of LOOPFROG

changed by the loop to unknown values. Additionally, a copy of the loop body is executed after the loop summary, such that assertions within the loop are preserved in the original context.

While this is already enough to prove some simple properties, much higher precision is required for more complex ones. As indicated in Fig. 4.1, LOOPFROG makes use of predefined abstract domains to achieve this. Every loop body of the model is passed to a set of abstract domains, through each of which a set of potential invariants of the loop is derived (heuristically).

The choice of the abstract domain for the loop summarization has a significant impact on the performance of the algorithm. A carefully selected domain generates fewer invariant candidates and thus speeds up the computation of a loop summary. Besides, the abstract domain has to be sufficiently expressive to retain enough of the semantics of the original loop to show the property.

Invariant candidates check

All potential invariants obtained from abstract domains always constitute an abstract (post-)state of the loop body, which may or may not be correct in the original program. To ascertain that a potential invariant is an actual invariant, LOOPFROG makes use of a verification engine. In the current version, the symbolic execution engine of CBMC [CKL04] is used. This engine allows for bit-precise, symbolic reasoning without abstraction. In our context, it always gives a definite answer, since only loop-less program fragments are passed to it. It is only necessary to construct an intermediate program that assumes the potential invariant to be true, executes the loop body once and then checks if the potential invariant still holds. If the verification engine returns a counterexample, we know that the potential invariant does not hold (i.e., it is not inductive); in the contrary case it must be a true invariant and it is subsequently added to the loop summary, since even after the program state is havoced, the invariant still holds. LOOPFROG starts this process from the innermost loop, and thus there is never an intermediate program that contains a loop. In case of nested loops, the inner loop is replaced with a summary, before the outer loop is analyzed. Due to this and the shortness of the fragments checked (only the loop body), small formulas are given to the verification engine and an answer is obtained quickly.

Verification of the resulting loop-less program abstraction

The result, after all loops have been summarized, is a loop-less abstraction of the input program. This abstract model is then handed to another verification engine. Again, the verification time is much lower than that of the original program, due to the model not containing any loops. As indicated by Fig. 4.1, the verification engine used to check the assertions in the abstract model, may be different from the one used to check potential invariants. In LOOPFROG, we choose to use the same engine (CBMC). We do so for two reasons: 1) it is very efficient and 2) it returns leaping counterexamples in case of assertion violations.

An abstract domain for analysis of string-manipulating programs

#	Constraint	Meaning
1	ZT_s	String s is zero-terminated
2	$L_s < B_s$	Length of s (L_s) is less than the size of the allocated buffer (B_s)
3	$0 \le i \le L_s$	Bounds on integer variables i (i is non-negative, i is bounded by buffer size, etc.) k is an arbitrary integer constant.
4	$0 \le i$	
5	$0 \le i < B_s$	
6	$0 \le i < B_s - k$	
7	$0 < \mathit{offset}(p) \le B_s$	Pointer offset bounds
8	$\mathit{valid}(p)$	Pointer p points to a valid object

Table 4.1. Examples of abstract domains tailored to buffer-overflow analysis.

The first experiments with LOOPFROG were done with a set of abstract domains that are specialized to buffer-related properties, in order to demonstrate the benefits of our approach on buffer-overflow benchmarks. The constrains of the domains are listed in Table 4.1.

We also make use of string-related abstract domains instrumented into the model similar to approach Dor et al. [DRS03]: for each string buffer s, a Boolean value z_s and integers l_s and b_s are tracked. The Boolean z_s holds if s contains the zero character within the buffer size b_s. If so, l_s is the index of the first zero character, otherwise, l_s has no meaning.

The chosen domains are instantiated according to variables occurring in the code fragment taken into account. To lower the amount of template instantiations, the following set of simple heuristics can be used:

1. Only variables of appropriate type are considered (we concentrate on string types).

2. Indices and string buffers are combined in one invariant only if they are used in the same expression, i.e., we detect instructions which contain $p[i]$ and build invariants that combine i with all string buffers pointed by p.

As shown in the next section these templates have proven to be effective in our experiments. Other applications likely require different abstract domains. However, new domain templates may be added quite easily: they usually can be implemented with less than a hundred lines of code. The example of implementation is provided in Appendix (Fig. A.1).

4.5 Experimental evaluation

In experiments we focus on ANSI-C programs: the extensive buffer manipulations in it provide a "ground" for multiple buffer overruns. We apply the domains from Table 4.1 to small programs collected in benchmarks suites and to real applications as well. All experimental data, an in-depth description of LOOPFROG, the tool itself, and all our benchmark files are available on-line for experimentation by other researchers[3].

All data was obtained on an 8-core Intel Xeon 3.0 GHz. We limited the run-time to 4 hours and the memory per process to 4 GB.

4.5.1 Evaluation on the benchmark suites

The experiments are performed on two recently published benchmark sets. The first one, by Zitser et al. [ZLL04], contains 164 instances of buffer overflow problems, extracted from the original source code of `sendmail`, `wu-ftpd`, and `bind`. The test cases do not contain complete programs, but only those parts required to trigger the buffer overflow. According to Zitser et al., this was necessary because the tools in their study were all either unable to parse the test code, or the analysis used disproportionate resources before terminating with an error ([ZLL04], pg. 99). In this set, 82 tests contain a buffer overflow, and the rest represent a fix of a buffer overflow.

We use metrics proposed by Zitser et al. [ZLL04] to evaluate and compare the precision of our implementation. We report the *detection rate* $R(d)$, i.e., percentage of correctly reported bugs, and the *false positive rate* $R(f)$ — percentage of incorrectly reported bugs in the fixed versions of the test cases. The *discrimination rate* $R(\neg f|d)$ is defined as the ratio of test cases on which an error is correctly reported, while it is, also correctly, not reported in the corresponding fixed test case. Using this measure, tools are penalized for not finding a bug, but also for not reporting a fixed program as safe.

The results of a comparison with a wide selection of static analysis tools[4] are summarized in Table 4.2. Almost all of the test cases involve array bounds violations. Even though Uno, Archer and BOON were designed to detect these type of bugs, they hardly report any errors. BOON abstracts all string manipulation using a pair of integers (number of allocated and used bytes) and does flow-insensitive symbolic analysis over collected constraints. These three

[3]`http://www.cprover.org/loopfrog/`

[4]The data for all tools but LOOPFROG, "$=$, $\neq$, $\leq$", and the Interval Domain is from [ZLL04].

tools use different approaches for the analysis. BOON and Archer do symbolic, while UNO does model-checking-based one. Archer and UNO are flow-sensitive, BOON is not. All three are interprocedural.

	$R(d)$	$R(f)$	$R(\neg f\|d)$
Loopfrog	1.00	0.38	0.62
$=, \neq, \leq$	1.00	0.44	0.56
Interval Domain	1.00	0.98	0.02
Polyspace	0.87	0.50	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	0
Archer	0.01	0	0
Uno	0	0	0
Loopfrog [KHCL07]	1.00	0.26	0.74
$=, \neq, \leq$[KHCL07]	1.00	0.46	0.54

Table 4.2. Effectiveness of various static analysis tool in Zitser et al. [ZLL04] and Ku et al. [KHCL07] benchmarks: detection rate $R(d)$, false positive rate $R(f)$, and discrimination rate $R(\neg f|d)$.

But they share the same problem — the approximation is too coarse and additional heuristics are applied in order to lower false positive rate, what all together makes hardly possible any complex bug to be detected. The source code of the test cases was not annotated, but nevertheless, the annotation-based Splint tool performs reasonably well on these benchmarks. Loopfrog and Interval Domain are the only entries that report all buffer overflows correctly (a detection rate of $R(d) = 1$) and with 62% Loopfrog also has the highest discrimination rate among all the tools. It is also worth noticing that our summarization technique performs quite well when only a few relational domains are used (the second line of Table 4.2). The third line in this table contains the data for a simple interval domain, not implemented in Loopfrog, but as a traditional abstract domain used in SatAbs model checker as a part of pre-processing; it reports almost everything as unsafe.

The second set of benchmarks was proposed by Ku et al. [KHCL07]. It contains 568 test cases, of which 261 are fixed versions of buffer overflows. This set partly overlaps with the first one, but contains source code of a greater variety of applications, including the Apache HTTP server, Samba, and the NetBSD C system library. Again, the test programs are stripped down, and are partly simplified to enable current model checkers to parse them. Our results on this set confirm the results obtained using the first set; the corresponding numbers are given in the last two lines of Table 4.2. On this set the advantage of selecting property-specific domains is clearly visible, as a 20% increase in the discrimination rate over the simple relational domains is witnessed. Also, the performance of Loopfrog is much better if specialized domains are used, simply because there are fewer candidates for the invariants.

				Time				Assertions		
Suite	**Program**	**Instructions**	**# Loops**	**Summari-zation**	**Checking Assertions**	**Total**	**Peak Memory**	**Total**	**Passed**	**Violated**
freecell-solver	aisleriot-board-2.8.12	347	26	10s	295s	305s	111MB	358	165	193
freecell-solver	gnome-board-2.8.12	208	8	0s	3s	4s	13MB	49	16	33
freecell-solver	microsoft-board-2.8.12	168	4	2s	9s	11s	32MB	45	19	26
freecell-solver	pi-ms-board-2.8.12	185	4	2s	10s	13s	33MB	53	27	26
gnupg	make-dns-cert-1.4.4	232	5	0s	0s	1s	9MB	12	5	7
gnupg	mk-tdata-1.4.4	117	1	0s	0s	0s	3MB	8	7	1
inn	encode-2.4.3	155	3	0s	2s	2s	6MB	88	66	22
inn	ninpaths-2.4.3	476	25	5s	40s	45s	49MB	96	47	49
ncompress	compress-4.2.4	806	12	45s	4060s	4106s	345MB	306	212	94
texinfo	ginstall-info-4.7	1265	46	21s	326s	347s	127MB	304	226	78
texinfo	makedoc-4.7	701	18	9s	6s	16s	28MB	55	33	22
texinfo	texindex-4.7	1341	44	415s	9336s	9757s	1021MB	604	496	108
wu-ftpd	ckconfig-2.5.0	135	0	0s	0s	0s	3MB	3	3	0
wu-ftpd	ckconfig-2.6.2	247	10	13s	43s	57s	27MB	53	10	43
wu-ftpd	ftpcount-2.5.0	379	13	10s	32s	42s	37MB	115	41	74
wu-ftpd	ftpcount-2.6.2	392	14	8s	24s	32s	39MB	118	42	76
wu-ftpd	ftprestart-2.6.2	372	23	48s	232s	280s	55MB	142	31	111
wu-ftpd	ftpshut-2.5.0	261	5	1s	9s	10s	13MB	83	29	54
wu-ftpd	ftpshut-2.6.2	503	26	27s	79s	106s	503MB	232	210	22
wu-ftpd	ftpwho-2.5.0	379	13	7s	23s	30s	37MB	115	41	74
wu-ftpd	ftpwho-2.6.2	392	14	8s	27s	35s	39MB	118	42	76
wu-ftpd	privatepw-2.6.2	353	9	4s	17s	22s	32MB	80	51	29

Table 4.3. Evaluation of LOOPFROG on the programs from wu-ftpd, texinfo, gnupg, inn, and freecell-solver tools suites.

The leaping counterexamples computed by our algorithm are a valuable aid in the design of new abstract domains that decrease the number of false positives. Also, we observe that both test sets include instances labeled as unsafe that LOOPFROG reports to be safe (1 in [ZLL04] and 9 in [KHCL07]). However, by manual inspection of the counterexamples for these cases, we find that our tool is correct, i.e., that the test cases are spurious.[5] For most of the test cases in the benchmark suites, the time and memory requirements of LOOPFROG are negligible. On average, a test case finishes within a minute.

4.5.2 Industrial code benchmarks

We also evaluated the performance of LOOPFROG on a real codebase, that is, complete un-modified program suites. Table 4.3 contains a selection of the results.

[5]We exclude those instances from our benchmarks.

These experiments clearly show that the algorithm scales reasonably well in both memory and time, depending on the program size and the number of loops contained. The time required for summarization naturally depends on the complexity of the program, but also to a large degree on the selection of (potential) invariants. As experience has shown, unwisely chosen invariant templates may generate many useless potential invariants, each requiring to be tested by the SAT-solver.

In general, the results regarding the program assertions shown to hold are not surprising; for many programs (e.g., texindex, ftpshut, ginstall), our selection of string-specific domains proved to be quite useful. It is also interesting to note that the results on the ftpshut program are very different on program versions 2.5.0 and 2.6.2: This program contains a number of known buffer-overflow problems in version 2.5.0, and considerable effort was spent on fixing it for the 2.6.2 release; an effort clearly reflected in our statistics. Just like in this benchmark, many of the failures reported by LOOPFROG correspond to known bugs and the leaping counterexamples we obtain allow us to analyze those faults. Merely for reference we list CVE-2001-1413 (a buffer overflow in ncompress) and CVE-2006-1168 (a buffer underflow in the same program), for which we are easily able to produce counterexamples.[6] On the other hand, some other programs (such as the ones from the freecell-solver suite) clearly require different abstract domains, suitable for other heap structures than strings. The development of suitable domains and subsequent experiments, however, are left for future research.

4.5.3 Comparison with the interval domain

To highlight the applicability of LOOPFROG to large-scale software and to demonstrate its main advantage, we present a comparative evaluation against a simple interval domain, which tracks the bounds of buffer index variables, an often employed static analysis. For this experiment, LOOPFROG was configured to use only two abstract domains, which capture the fact that an index is within the buffer bounds (#4 and #5 in Table 4.1). As apparent from Table 4.4, the performance of LOOPFROG in this experiment is far superior to that of the simple static analysis.

We analyze a single benchmark in detail, in order to explain the data delivered by the tool: The `ncompress` program (version 4.2.4) contains about 2.2K lines of code (which translates to 963 instructions in the model file) and 12

[6]The corresponding bug reports may be obtained from `http://cve.mitre.org/`.

			LOOPFROG		Interval Domain	
Suite	Benchmark	Total	Failed	Ratio	Failed	Ratio
bchunk	bchunk	96	8	0.08	34	0.35
freecell-solver	make-gnome-freecell	145	40	0.28	140	0.97
freecell-solver	make-microsoft-freecell	61	30	0.49	58	0.95
freecell-solver	pi-make-microsoft-freecell	65	30	0.46	58	0.89
gnupg	make-dns-cert	19	5	0.26	19	1.00
gnupg	mk-tdata	6	0	0.00	6	1.00
inn	encode	42	11	0.26	38	0.90
inn	ninpaths	56	19	0.34	42	0.75
ncompress	compress	204	38	0.19	167	0.82
texinfo	makedoc	83	46	0.55	83	1.00
wu-ftpd	ckconfig	1	1	1.00	1	1.00
wu-ftpd	ftpcount	61	7	0.11	47	0.77
wu-ftpd	ftpshut	63	13	0.21	63	1.00
wu-ftpd	ftpwho	61	7	0.11	47	0.77

Table 4.4. Comparison between LOOPFROG and an interval domain: The column labelled 'Total' indicates the number of properties in the program, and 'Failed' shows how many of the properties were reported as failing; 'Ratio' is Failed/Total.

loops. During preprocessing, LOOPFROG detected 204 potential buffer overflows and inserted an assertion for each of them in the model. Loop summarization took 14.4 seconds. During this time, 67 potential invariants were created and 17 of them were confirmed as actual invariants. The overall analysis took 668 seconds. Finally, 166 assertions hold and 38 are reported as failing (while producing leaping counterexamples for each violation).

To evaluate scalability, we applied other verification techniques to this example. CBMC [CKL04] tries to unwind all the loops, but fails, reaching the 2GB memory limit. The same behavior is observed using SATABS [CKSY05b], where the underlying model checker (SMV) hits the memory limit.

4.6 Related work and summary

Other work on analysis using summaries of functions is quite extensive (see a nice survey in [GR07]) and dates back to Cousot and Halbwachs [CH78b], and Sharir and Pnueli [SP81]. In a lot of projects, function summaries are created for alias analysis or points-to analysis, or are intended for the analysis of program fragments. As a result, these algorithms are either specialized to particular problems and deal with fairly simple abstract domains or are restricted to analysis of parts of the program. An instance is the summarization of library functions in [GR07]. In contrast, our technique aims at computing a summary for the entire program, and is applicable to complex abstract domains.

The same practical motivation, sound analysis of ANSI-C programs, drives our work and the work behind Frama-C project [Fra]. In particular, the PhD work of Moy [Moy09] even targets, among others, the same set of benchmarks — Verisec [KHCL07] and Zitser's [ZLL04] test suites. To tackle them with Frama-C tools Moy employs a number of techniques that discover pre- and post-conditions for loops as well as loop invariants. He combines abstract interpretation-based invariant inference with weakest precondition-based iterative methods such as Suzuki-Ishihata algorithm [SI77]. The latter one, induction iteration, applies weakest precondition computation to a candidate loop invariant iteratively until inductive invariant is found. Thus, loop summarization can be seen as 1-step application of the induction-iteration method, in which weakest precondition computation is replaced with strongest postcondition one[7].

Note that application of the Suzuki-Ishihata algorithm to string operation-intensive programs (as our benchmarks are) often leads to non-terminating iterative computation since there is no guarantee to obtain an inductive invariant from a candidate. To avoid this uncertainty, we are interested only in those candidates that can be proven to be an inductive invariant in a single step. We claim that a careful choice of candidates would contribute more to precision and scalability of analysis. In fact, our results on the aforementioned benchmark suites support the claim. We analyze Zitser's benchmark suite in a matter of seconds and are able to discharge 62% of bug-free instances, while Frama-C does not complete any of test cases within 1 hour limit. When applied to a smaller programs of the Verisec test suite both tools are able to discharge 74% of bug-free test cases; LOOPFROG required almost no time for this analysis.

LOOPFROG shares a lot in idea and architecture with Houdini, an annotation assistant for ESC/Java [FL01]. Houdini was first created as a helper to ESC/-Java; the goal was to lower the burden of manual program annotation (having enough useful annotations is critical for successful application of ESC/Java). Similar to loop summarization, Houdini "magically" guesses a set of candidate relations between program variables and then discharges or verifies them one by one using the ESC/Java as a refuter. Verified candidates are added to the program as annotations and are used later by the main ESC/Java check of a program in a same way as symbolic execution makes uses of summaries when

[7]However, the choice of inference"direction", i.e. pre-condition or post-condition computation, is irrelevant for 1-step application.

it runs over a loop-free program.

However there are also numerous differences between these two. Houdini is designed to be applied to any program module or a routine in a library while our summarization concentrates deliberately on loops. Houdini adds annotations to the program, while LOOPFROG replaces each loop with the summary, thus keeping the cost of analysis for every consecutive loop as low as was the inner-most one.

Houdini as well as LOOPFROG suggest a lot of candidates that help to address buffer access checks. For instance, it generates 6 different comparison relations for each integral type and a constant in a program. However, while experimenting with LOOPFROG, we found such an abstract domain of arbitrary relations to be effective, though very expensive. With its help we generate too many candidates and find too many useless loop invariants. Therefore we prefer problem-tailored domains that generate fewer candidates, but keep the "usefulness" at an appropriate level.

Also, as we show later in Chapter 5, LOOPFROG extends candidates selection to those that relate two different valuations of the same program variable, e.g., before and after a loop iteration. This allows discovering not only safety, but also liveness-related loop invariants; in particular loop termination can be proven with the help of this addition.

A series of work by Gulwani et al. [GLAS09, GMC09] uses loop invariants discovery for the purpose of worst-case execution time analysis. One of the approaches (reported as practically the most effective one) employs template-based generation of invariant candidates. Starting from the inner-most loop, a bound of the loop's maximal resources usage is computed. Therefore, it can be seen as a loop summarization with the domains tuned for WCET-analysis rather then string-operations as in the current LOOPFROG.

The Saturn tool [ABD$^+$07] computes a summary of a function with respect to an abstract domain using a SAT-based approach to improve scalability. However, summaries of loop-bodies are not created. In favor of scalability, Saturn simply unwinds loops a constant number of times, and thus, is unsound as bugs that require more iterations are missed.

SAT-solvers, SAT-based decision procedures, and constraint solvers are frequently applied in program verification. Notable instances are Jackson's Alloy tool [JV00] and CBMC [CKL04]. The SAT-based approach is also suitable for computing abstractions, as, for example, in [ABD$^+$07, CKSY04, RSY04] (see detailed discussion in Sec. 4.2.1). The technique reported in this Chapter also uses the flexibility of a SAT-based decision procedure for a combination of theories to compute loop summaries.

Our technique can be used for checking buffer overruns and class-string vulnerabilities. There exist a large number of static analysis tools focusing on these particular problems (a summary is given in Section 4.5). In this respect, the principal difference of our technique is that it is a general purpose abstraction-based checker which is not limited to special classes of faults.

A major benefit of our approach is its ability to generate diagnostic information for failed properties. This is usually considered a distinguishing feature of *model checking* [CGP99] and, sometimes, extended static checking [FLL$^+$02], but rarely found in tools based on abstract interpretation. Most model checkers for programs implement a CEGAR approach [BR01, HJMS02], which combines model checking with counterexample-guided abstraction refinement. The best-known instance is SLAM [BR01], and other implementations are BLAST [HJMS02], MAGIC [CCG$^+$04], and SatAbs [CKSY05b], which implement predicate abstraction. Recently, a number of projects applied counterexample-guided refinement to abstract domains other than predicate abstraction. Manevich et al. [MFH$^+$06] formalize CEGAR for general powerset domains; Beyer et al. [BHT06] integrate the TVLA system [LAS00] into BLAST and use counterexamples to refine 3-valued structures to make shape analysis more scalable; Gulavani and Rajamani devised an algorithm for refining any abstract interpretations [GR06, GCNR10] by combining widening with interpolation. Our procedure is also able to generate counterexamples with respect to the abstract domain and could be integrated into a CEGAR loop for automatic refinement.

Summary

We presented a novel algorithm for program verification using symbolic abstract transformers. The algorithm computes an abstraction of a program with respect to a given abstract interpretation by replacing loops and function calls in the control flow graph by their symbolic transformers. The run-time of our algorithm is linear in the number of looping constructs in the program. It addresses the perennial problem of the high complexity of computing abstract fixpoints. The procedure over-approximates the original program, which implies soundness of our analysis. An additional benefit of the technique is its ability to generate *leaping counterexamples*, which are helpful for diagnosis of the error or for filtering spurious warnings. Experimental results show the best error-detection and error-discrimination rates comparing to a broad selection of static analysis tools. The implemented Loopfrog tool is available for experiments by other researchers.

Chapter 5

Termination Proofs via Loop Summarization

I hate assumptions. Assumption is
the mother of all mess-ups.

Brian Sheehy

This chapter proposes a new approach to establishing program termination proofs. We employ abstract domains, capable of encoding transition invariants, for loop summarization. Transitivity (or compositionality) is used as a criterion that ensures transition invariants are strong enough to conclude loop termination.

5.1 Introduction

The problem of program termination (also known as the uniform halting problem) is one of the oldest and most well-known in the arena of computer science. Rooted in the historical Hilbert's *Entscheidungsproblem* it can be formulated as follows:

> *In finite time, determine whether a given program always finish running or could execute forever.*

Turing is famous for showing its undecidability [Tur36]. The publicity plays a bad trick with the problem — its formulation is, sometimes, exaggerated to a level that people believe termination for any program cannot be proven. In contrast to this popular belief, it can be proven for many realistic classes of programs. In fact, the study of program termination received increased interest in the recent past. Termination analysis is at a point where industrial application of termination proving tools is feasible. This is possible through a series of improvements upon methods introduced by the same Turing [Tur49]. The key fact underlying these methods is that termination may be reduced to the construction of *well-founded ranking relations*. Such a relation establishes an order between the states of the program, or, to say it differently, it ranks all the states — assigns them with natural numbers such that for any pair of consecutive states s_i, s_{i+1} the rank is decreasing, i.e., $rank(s_{i+1}) < rank(s_i)$. The existence of such an assignment ensures well-foundedness of the given set of transitions. Consequently, a program is terminating if there exists a *ranking function* for every program execution.

Podelski and Rybalchenko proposed *disjunctive* well-foundedness of transition invariants [PR04b](defined in Chapter 2.2.1) as a means to improve the performance of termination proving, as well as to simplify synthesis of ranking relations. Based on their crucial discovery, the same authors together with Cook gave an algorithm to verify program termination using iterative construction of transition invariants — the *Terminator* algorithm [CPR05, CPR06]. This algorithm exploits the relative simplicity of ranking relations for a single path of a program. It relies on a safety checker to find previously unranked paths of a program, computes a ranking relation for each of them individually, and disjunctively combines them in a global (disjunctively well-founded) termination argument. This strategy shifts the complexity of the problem from ranking relation synthesis to safety checking, a problem for which many efficient solutions exist (mainly, by means of model cheking-based reachability analysis).

The Terminator algorithm was successfully implemented in tools (e.g., TERMINATOR [CPR06], ARMC [PR07], SATABS [CKRW10]) and applied to verify industrial code, most notably, Windows device drivers. However, it has subsequently become apparent that the safety check is a bottleneck of the algorithm, taking up to 99% of the run-time [CPR06, CKRW10] in practice. The runtime required for ranking relation synthesis is negligible in comparison. A solution to this performance issue is *Compositional Termination Analysis (CTA)* [KSTW10]. This method limits path exploration to several iterations of each loop of the program. Transitivity (or *compositionality*) of the intermediate ranking arguments is used as a criterion to determine when to stop the loop unwinding. This allows for a reduction in runtime, but introduces incompleteness since a transitive termination argument may not be found for each loop of a program. However experimental evaluation on Windows device drivers confirmed that this case is rare in practice.

The complexity of the termination problem together with the observation that most loops in practice have (relatively) simple termination arguments suggests the use of light-weight static analysis for this purpose. In particular, this Chapter proposes a termination analysis based on the loop summarization algorithm described in Chapter 4. We build a new technique for termination analysis by 1) employing an abstract domain of (disjunctively well-founded) transition invariants during summarization and 2) using compositionality check as a completeness criterion for discovered transition invariant.

As before, our algorithm constructs summaries for loops, starting from the inner-most loop in the control flow graph of the program. In case of nested loops, inner loops are replaced with their summaries during verification. At any point during the analysis, the problem is therefore reduced to the analysis of a single loop. During construction of the loop summaries, our algorithm relies on a library of templates for abstract domains. These are used to construct candidates for transition invariants which subsequently are verified to be actual disjunctively well-founded transition invariants by means of a safety checker and a satisfiability decision procedure. Due to the fact that the safety checker is employed to analyze only a single unwinding of a loop at any point, we gain large speedups compared to algorithms like Terminator or CTA. At the same time, the false-positive rate of our algorithm is very low in practice, which we demonstrate in an experimental evaluation of our algorithm on a large set of Windows device drivers.

The rest of the Chapter is organized as follows: Section 5.2 introduces required theoretical concepts and, in particular, demonstrates the flow from TERMINATOR via CTA to our new algorithm. Section 5.3 presents our new method. Section 5.4 proposes an optimization that simplifies the selection of candidates for transition invariants. In Section 5.5 we give experimental evidence of the practicality of our approach and, finally, Section 5.6 discusses the related work and summarizes.

5.2 Preliminaries

We consider programs as transition systems $P = \langle S, I, R\rangle$ (Definition 1, page 13) and rely on the notion of (disjunctively well-founded) transition invariants introduced by Podelski and Rybalchenko [PR04b]. It was stated in this work in Chapter 2, but we repeat it here for convenience of the reader:

Definition 9 (Transition Invariant [PR04b]). *A* transition invariant *T for program P represented by a transition system $\langle S, I, R\rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.*

Definition 8 (Disjunctive Well-foundedness [PR04b]). *A relation T is* disjunctively well-founded (d.wf.) *if it is a finite union $T = T_1 \cup \cdots \cup T_n$ of well-founded relations.*

The main result of the work [PR04b] concludes program termination from the existence of disjunctively well-founded transition invariant.

Theorem 1 (Termination [PR04b]). *A program P is terminating iff there exists a d.wf. transition invariant for P.*

The result was put in use by the Terminator[1] algorithm [CPR06] that automates construction of d.wf transition invariants. It starts with an empty termination condition $T = \emptyset$ and queries a safety checker for a counterexample — a computation that is not covered by the current termination condition T. Next, a ranking relation synthesis algorithm is used to obtain a termination argument T' covering the transitions in the counterexample. The termination argument is then updated as in $T = T \cup T'$ and the algorithm continues to query for counterexamples. Finally, either a complete (d.wf.) transition invariant is constructed or there does not exist a ranking relation for some counterexample, in which case the program is reported as non-terminating.

5.2.1 Compositional termination analysis

Podelski and Rybalchenko [PR04b] remarked an interesting fact regarding the compositionality (transitivity) of transition invariants: If T is transitive, it is enough to show that $T \supseteq R$ instead of $T \supseteq R^+$ to conclude termination, because a compositional and d.wf. transition invariant is well-founded, since it is an inductive transition invariant for itself [PR04b]. Therefore, compositionality of a d.wf. transition invariant implies program termination.

To comply with the terminology in the existing literature, we define the notion of compositionality for transition invariants as follows:

Definition 18 (Compositional Transition Invariant [PR04b, KSTW10]). *A d.wf. transition invariant T is called* compositional *if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.*

[1]The Terminator algorithm is, sometimes, referred to as Binary Reachability Analysis (BRA), though BRA is only a particular technique to implement the algorithm (e.g., [CKRW10]).

This useful property did not find its application in transition invariant-based termination analysis until 2010. To understand its value we need to look closer at the transitive closure of a program's transition relation R. The safety checker in the Terminator algorithm verifies that a candidate transition invariant T indeed includes R^+ restricted to the reachable states. Note that the (non-reflexive) transitive closure of R is essentially an unwinding of program loops:

$$R^+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{i=1}^{\infty} R^i \,.$$

Thus, instead of searching for a d.wf. transition invariant that is a superset of R^+, we can therefore decompose the problem into a series of smaller ones. We can consider a series of loop-free programs in which R is unwound k times, i.e., the program that contains the transitions in $R^1 \cup \ldots \cup R^k$. As was shown in [KSTW10], if there is a d.wf. T_k with $\bigcup_{j=1}^{k} R^j \subseteq T_k$ and T_k is also transitive, then T_k is a compositional transition invariant for P.

This idea results in an algorithm that constructs d.wf. relations T_i for incrementally deep unwindings of P until it finally finds a transitive T_k, which proves termination of P. In [KSTW10], the algorithm was named a *Compositional Termination Analysis* (CTA); it is depicted in Algorithm 5.

This algorithm makes use of an external ranking procedure called $rank$, which generates a d.wf. ranking relation for a given set of transitions, or alternatively a set $C \in S$ of states such that $R^*(C)$ contains infinite computations. We say that $rank$ is sound if it always returns either a d.wf. superset of its input or a non-empty set of states C, and we call it complete if it terminates on every input.

The literature presents a broad range of methods to implement a $rank$ procedure. Usually, this is accomplished via synthesis of *ranking functions*, which define well-founded *ranking relations* [CKRW10, CS01, PR04a, BMS05]. Both Terminator and CTA are making use of such a ranking synthesis.

Algorithm 5 maintains a set $X \subseteq S$ that is an over-approximation of the set of reachable states, i.e., $R^*(I) \subseteq X$. It starts with $X = S$ and $i = 1$. It iterates over i and generates d.wf. ranking relations T_i for the transitions in $\bigcup_{j=1}^{i} R^j \setminus T$. As long as such relations are found, they are added to T. Once it finds a transitive T, the algorithm stops, as P is proven to terminate. When ranking fails for some i, the algorithm checks whether there is a reachable state in C, in which case $R^*(C)$ contains a counterexample to termination and the algorithm consequently reports P as non-terminating. Otherwise, it removes C from X, which represents a refinement of the current over-approximation of

Algorithm 5: Compositional Termination Analysis [KSTW10]

```
input  : $P = \langle S, I, R \rangle$
output: 'Terminating' / 'Non-Terminating'
begin
    $T := \emptyset$;
    $X := S$;
    $i := 1$;
    while true do
        $\langle T_i, C \rangle := rank\ ((\bigcup_{j=1}^{i} R^j \setminus T) \cap (X \times X))$;
        if $C \cap R^*(I) \neq \emptyset$ then
            return 'Non-Terminating';
        else if $C = \emptyset$ and $T \cup T_i$ is transitive then
            return 'Terminating';
        else
            $X := X \setminus C$;
            $T := T \cup T_i$;
            $i := i + j$, where $j > 0$;
        end if
    end while
end
```

the set of reachable states.

Lines 12–14 ensure progress between iterations by excluding unreachable states (C) from the approximation X and adding the most recently found T_i in T. However, for non-terminating input programs, the algorithm may not terminate for two reasons: a) $rank$ is not required to terminate, and b) there may be an infinite sequence of iterations. This is not the case for finite S if the input program is non-terminating, since sound and complete ranking procedures exist (e.g., [PR04b, CKRW10]) and progress towards the goal can thus be ensured.

From Terminator via CTA to a light-weight static analysis

Terminator is a complete algorithm (with regards to completeness of the ranking procedure). Note that CTA is not complete for terminating programs even if they are finite-state. This is due to the fact that T is not guaranteed to ever become transitive, even if it contains R^+.

Terminator strategy can be seen as a "proof by construction": it explicitly builds the valid terminating argument for every path in a program. CTA combines “proof by construction” with a “proof by induction”: it first tries to construct a base step and then checks the inductiveness of it. Surely, not for every problem an inductive proof exists. However, as it was shown in [KSTW10] for loops in industrial applications such as Windows device drivers, the CTA approach pays off with several orders of improvement over TERMINATOR.

That gives us an intuition to go further in lightening the proof strategy — from a mix of “proof by construction” with a “proof by induction” to a pure “proof by induction”. I.e., a technique where ranking synthesis-based transition invariant discovery is replaced by abstract domain-based transition invariant discovery. Instead of a complex base case construction, we “guess” several variants of it using lightweight static analysis methods and then check if the inductive step holds as well. The method is still incomplete, but allows avoiding non-effective path enumeration inside the safety checker[2].

We propose to use for this purpose the loop summarization (described in Chapter 4) with the specific relational domains. We show the details of it later in Section 5.3, but first we formulate several useful summarization subroutines.

5.2.2 Loop summarization subroutines

First we recap the summarization algorithm of Chapter 4 such that it conforms to the program modelling used for termination analysis. Algorithm 6 presents an outline of the procedure. The function SUMMARIZE traverses the control-flow graph of the program P and calls itself recursively for each block with nested loops. If a block contains a non-nested loop, it is summarized using the function SUMMARIZELOOP and the resulting summary replaces the original loop in P'. Thereby, outer loops become non-nested as well, which enables further progress.

The function SUMMARIZELOOP computes the summaries. The most general abstraction is to replace a loop by a program fragment that ‘havocs’ the state by setting all variables which are (potentially) written to during loop execution to non-deterministic values. To improve the precision of these summaries, they are strengthened by (partial) loop invariants. SUMMARIZELOOP has two subroutines that are related to invariant discovery: PICKINVARIANTCANDIDATES, which returns a set of ‘invariant candidates’ depending on an abstract interpretation

[2]As a future stand-alone theoretical problem we see a definition of a class of systems and properties, for which termination (liveness) can be proved by induction, i.e., by guess and proof of the base step (discovery of a transition invariant) and proof of the inductive step (compositionality of a transition invariant).

Algorithm 6: Routines of loop summarization (details are in Chapter 4)

```
SUMMARIZE($P$)
input: program $P$
output: Program summary
begin
    foreach Block $B$ in CONTROLFLOWGRAPH($P$) do
        if $B$ has nested loops then
            $B$ :=SUMMARIZE($B$)
        else if $B$ is a single loop then
            $B$ :=SUMMARIZELOOP($B$)
        endif
    end foreach
    return $P$
end

SUMMARIZELOOP($L$)
input: Single-loop program $L$ (over variable set $X$)
output: Loop summary
begin
    $I := \top$
    foreach Candidate $C(X)$ in PICKINVARIANTCANDIDATES($L$) do
        if ISINVARIANT($L$, $C$) then
            $I := I \wedge C$
        endif
    end foreach
    return "$X^{pre} := X$; havoc($X$); assume($I(X^{pre}) \implies I(X)$);"
end

ISINVARIANT($L$, $C$)
input: Single-loop program $L$ (over variable set $X$), invariant candidate
  $C$
output: TRUE if $C$ is invariant for $L$; FALSE otherwise
begin
    return UNSAT($\neg(L(X,X') \wedge C(X) \Rightarrow C(X'))$)
end
```

based static analysis of the loop and IsInvariant that established whether a candidate is an actual invariant.

Note that this summarization algorithm does not take loop termination into account. The summaries computed by the algorithm are always terminating program fragments. The abstraction is therefore a sound over-approximation, but it may be too coarse for programs that contain unreachable paths.

5.3 Loop summarization with transition invariants

In this section we introduce a method that allows *transition* invariants to be included for strengthening of loop summaries. This increases the precision of the summaries by allowing loop termination to be taken into account.

According to Definition 9 (page 20), a binary relation T is a transition invariant for a program P if it contains R^+ (restricted to the reachable states). Note, however that transitivity of T is also a sufficient condition when T is only a superset of R:

Theorem 6. *A binary relation T is a transition invariant for the program $\langle S,I,R\rangle$ if it is transitive and $R \subseteq T$.*

Proof. From transitivity of T it follows that $T^+ \subseteq T$. Since $R \subseteq T$ it follows that $R^+ \subseteq T$. □

This simple fact allows for an integration of transition invariants into the loop summarization framework by a few adjustments to the original algorithm. Consider line 19 of Algorithm 6, where candidate invariants are selected. Clearly, we need to allow selection of transition invariants here, i.e., invariant candidates now take the form $C(X,X')$, where X is the post-state of L.

What follows is a check for invariance of C over $L(X,X')$, i.e., a single unwinding of the loop. Consider the temporary (sub-)program $\langle S,S,L\rangle$ to represent the execution of the loop from a non-deterministic entry state, as required by IsInvariant. A transition invariant for this program is required to cover L^+, which, according to Theorem 6, is implied by $L \subseteq C$ and transitivity of C. The original invariant check in IsInvariant establishes $L \subseteq C$, when the check for unsatisfiability receives the more general formula $L(X,X') \wedge C(X,X')$ as a parameter. The summarization procedure furthermore requires a slight change to include a check for compositionality. The resulting procedure is Algorithm 7.

Algorithm 7: Loop summarization with transition invariants.

```
SUMMARIZELOOP-TI($L$)
input: Single-loop program $L$ with a set of variables $X$
output: Loop summary
begin
    $T := \top$
    foreach Candidate $C(X,X')$ in PICKINVARIANTCANDIDATES($L$) do
        if ISINVARIANT($L$, $C$) $\wedge$ ISCOMPOSITIONAL($C$) then
            $T := T \wedge C$
        endif
    end foreach
    return "$X^{pre} := X$; havoc($X$); assume($T(X^{pre},X)$);"
end
```

The additional compositionality (transitivity) check ISCOMPOSITIONAL at line 7 of Algorithm 7 corresponds to a check for unsatisfiability of

$$\exists s_i, s_j, s_k \in S \,.\, \neg \left(C(s_i,s_j) \wedge C(s_j,s_k) \Rightarrow C(s_i,s_k) \right) \ , \tag{5.1}$$

which may be decided by a suitable decision procedure, e.g., SAT- or SMT-solver. Formula 5.1 has only existential quantification and, thus, its complexity for state-of-the-art decision procedures depends only on the complexity of the candidate relation C. If no quantified logic is used for a expressing C, Fomula 5.1 can be encoded into a propositional or a decidable first-order problem.

Of course, this check may be omitted if the selected invariant candidates are compositional by construction.

Termination

The changes to the summarization algorithm allow for termination checks during summarization through application of Theorem 1, which requires a transition invariant to be disjunctively well-founded. This property may be established by allowing only disjunctively well-founded invariant candidates, or it may be checked by means of decision procedures (e.g., SMT solvers where applicable).

According to Definition 8 (page 20) in order to ensure d.wf. of a candidate relation T the well-foundedness of each of its disjuncts must be established.

This can be done by explicit encoding of the well-foundedness criteria of Definition 7 (page 20). However the resulting formula contains quantifiers. The consequence is that, in the case of infinite state systems, decision procedures may be unable to decide it.

The difference with TERMINATOR and CTA

The complexity of establishing well-foundedness of a transition invariant hints at the explanation of a major difference between our new algorithm and TERMINATOR/CTA. The latter two construct the transition invariant using the abstraction-refinement loop such that it is already disjunctively well-founded, while we allow any transition invariant to be discovered, though, later it needs to be checked for well-foundedness. Note that even if the discovered transition invariant is not well-founded, it is still a valid part of a loop summary, i.e., it can be used to improve precision of the constructed abstraction.

However, the research in size-change termination for functional languages [BAL09][3] suggests that a small set of templates for ranking relations is enough to cover many classes of programs. Besides, the expensive well-foundedness check can be completely omitted if we employ specialized abstract domains that produce only well-founded candidates for transition invariants. We propose a solution for that in Section 5.4.

Example 5.1. *Consider the program in Figure 5.1. The symbolic transformer for the loop body is:* $\phi_L := x' = x + 1$. *Also consider the relation "$>$" for a pair x' and x as a candidate relation T_c. T_c is indeed a transition invariant if the following formula is unsatisfiable:*

```
int x = 0;
while(x<255)
  x++;
```

Figure 5.1. An example of a terminating loop with a strictly increasing iterator

$$x < 255 \wedge x' = x + 1 \wedge \neg(x' > x)\,.$$

The formula is UNSAT, the invariant holds, and $x' > x$ is added to the symbolic transformer as a transition invariant. Since the relation is compositional and d.wf. (we show later why) the loop is marked as terminating.

[3]We discuss the relation of our method to size-change termination later in Section 5.6.1

5.4 Invariant candidate selection

In this section, we propose a set of specialized candidate relations which we find to be useful in practice, as demonstrated in the following section. We focus on transition invariants for machine-level integers (i.e., finite integers with overflows) for a bit-precise analysis of programs implemented in low-level languages like ANSI-C.

In contrast to other work on termination proving with abstract domains (e.g., [BCC⁺07]), we do not aim for general domains like Octagon and Polyhedra. Although fast in computation, they are not designed for termination and d.wf. and compositionality checks for them can be costly. Instead we focus on domains that

- generate few, relatively simple candidate relations;
- allow for efficient d.wf. and compositionality checks.

Arithmetic operations on machine-level integers usually allow overflows, e.g., the instruction $i = i+1$ for a pre-state $i = 2^k - 1$ results in a post-state $i' = -2^{k-1}$ (when represented in two's-complement). If termination of the loop depends on machine-level integers, establishing well-foundedness of a relation over it is not straightforward — monotonical increase/decrease for this set of numbers can be affected by overflow. However we can use the following theorem to simplify the discovery of a d.wf. transition invariant.

Theorem 7. *If $T : K \times K$ is a strict order relation for a finite set $K \subseteq S$ and is a transition invariant for the program $\langle S, I, R\rangle$, then T is well-founded.*

Proof. If T is a transition invariant, than it holds also for all pairs $(k_1, k_2) \in K \times K$. Thus, it is total over K. Non-empty finite totally-ordered sets always have a least element and, therefore, T is well-founded. □

The proof uses the fact that, when checking T for being a transition invariant, we implictely enumerated all the pairs of pre- and post-states to discover if any of them violates the order.

A total strict order relation is also transitive, which allows for an alternative (stronger) criterion than Theorem 1:

Corollary 2. *A program terminates if it has a transition invariant T that is also a finite strict order relation.*

#	Constraint	Meaning
1	$i' < i$ $i' > i$	A numeric variable i is strictly decreasing (increasing).
2	$x' < x$ $x' > x$	Any loop variable x is strictly decreasing (increasing) in lexicographical order.
3	$sum(x', y') < sum(x, y)$ $sum(x', y') > sum(x, y)$	Sum of all numeric loop variables is strictly decreasing (increasing).
4	$max(x', y') < max(x, y)$ $max(x', y') > max(x, y)$ $min(x', y') < min(x, y)$ $min(x', y') > min(x, y)$	Maximum or minimum of all numeric loop variables is strictly decreasing (increasing).
5	$(x' < x \wedge y' = y) \vee$ $(x' > x \wedge y' = y) \vee$ $(y' < y \wedge x' = x) \vee$ $(y' > y \wedge x' = x)$	A combination of strict increase or decrease for one of loop variables while the remaining ones are not updated.

Table 5.1. Templates of abstract domains used to draw transition invariant candidates

This corollary allows for a selection of invariant candidates that ensures (disjunctive) well-foundedness of transition invariants. An explicit check is therefore not required. An example of such a candidate appears in Example 5.1

Note that strictly ordered and finite transition invariants exist for many programs in practice: machine-level integers or strings of fixed length have a finite number of possible distinct pairs and strict natural or lexicographical orders are defined for them as well.

5.5 Evaluation

For a proof of concept we put the described theory in use in our static analyzer Loopfrog (Section 4.4). As before, the tool works with the program models produced by Goto-CC model extractor; ANSI-C programs are considered as a primary experimental target.

We implemented a number of domains based on strict order numeric relations, thus, following Corollary 2, additional checks for compositionality and d.wf-ness of candidate relations were not required. The domains are listed in

Table 5.1. Here we report the results for two most illustrative schemata:

- LOOPFROG 1: domain #3 in Table 5.1. Expresses the fact that a sum of all numeric variables of a loop is strictly decreasing (increasing). Fastest approach, because it generates very few (but large) invariant candidates per loop;

- LOOPFROG 2: domain #1 in Table 5.1. Expresses strict decrease (increase) for every numeric variable of a loop. Generates twice as many simple strict order relations as there are variables in a loop;

As a reference tool we used a termination prover built upon the CBMC-SATABS [CKSY05b] framework. This tool implements Compositional Termination Analysis (CTA) [KSTW10] and Binary Reachability Analysis of TERMINATOR algorithm [CPR05]. For both the default ranking function synthesis methods were enabled — templates for relations on bit-vectors with SAT-based enumeration of coefficients; for more details see [CKRW10].

We experimented with a large number of ANSI-C programs including:

- The SNU real-time benchmark suite that contains small C programs used for worst-case execution time analysis [snu];

- The Powerstone benchmark suite as an example set of C programs for embedded systems [SLC+99];

- The Verisec 0.2 benchmark suite [KHCL07];

- The Jhead 2.6 utility;

- The Bchunk 1.2.0 utility;

- Windows device drivers (from Windows Device Driver Kit 6.0).

All experiments were run on an Ubuntu server equipped with Dual-Core 2GHz Opteron 2212 CPU and 4GB of memory. The analysis was set to run with the timeout of 120 minutes for all loops at once (LOOPFROG) or of 60 minutes per loop (CTA and TERMINATOR).

The results for SNU, Powerstone, Bchunk and Jhead are presented in Tables 5.3, 5.4, 5.6 and 5.5. Each table in columns 3 to 5 reports quantity of loops that were proven as terminating (T), potentially non-terminating (NT) and time-out (TO) for each of the compared techniques.

Time in column 6 is computed only for loops noted in T and NT; time-outed loops are not included in total time. Instead, '+' is used to denote the cases where at least one time-out occured.

The results for the Verisec 0.2 benchmark suite are given in the aggregated form in Table 5.2. The suite consists of a large number of stripped C programs that correspond to known security bugs. Although each program has very few loops, the variety of loop types is fairly big and, thus, is interesting for analysis.

	Method	**T**	**NT**	**TO**	**Time**	
244 loops in 160 benchmarks	Loopfrog 1	33	211	0	11.38	
	Loopfrog 2	44	200	0	22.49	
	CTA	34	208	2	1207.62	+
	Terminator	40	204	0	4040.53	

Table 5.2. Aggregated data of comparison between Loopfrog, CTA and Terminator on the Verisec 0.2 suite.

Benchmark	**Method**	**T**	**NT**	**TO**	**Time**	
adpcm-test 18 loops	Loopfrog 1	13	5	0	470.05	
	Loopfrog 2	17	1	0	644.09	
	CTA	13	3	2	260.98	+
	Terminator	12	2	4	165.67	+
bs 1 loop	Loopfrog 1	0	1	0	0.05	
	Loopfrog 2	0	1	0	0.12	
	CTA	0	1	0	12.22	
	Terminator	0	1	0	18.47	
crc 3 loops	Loopfrog 1	1	2	0	0.17	
	Loopfrog 2	2	1	0	0.26	
	CTA	1	1	1	0.21	+
	Terminator	2	1	0	13.88	
fft1k 7 loops	Loopfrog 1	2	5	0	0.36	
	Loopfrog 2	5	2	0	0.67	
	CTA	5	2	0	141.18	
	Terminator	5	2	0	116.81	
fft1 11 loops	Loopfrog 1	3	8	0	3.68	
	Loopfrog 2	7	4	0	4.98	
	CTA	7	4	0	441.94	
	Terminator	7	4	0	427.36	
fir 8 loops	Loopfrog 1	2	6	0	2.90	
	Loopfrog 2	6	2	0	8.48	
	CTA	6	2	0	2817.08	
	Terminator	6	1	1	236.70	+

Continued on the next page…

Table 5.3. SNU real-time benchmarks suite: comparison of Loopfrog with CTA and Terminator.

Table 5.3 – Continued

Benchmark	Method	T	NT	TO	Time
insertsort 2 loops	Loopfrog 1	0	2	0	0.05
	Loopfrog 2	1	1	0	0.06
	CTA	1	1	0	226.45
	Terminator	1	1	0	209.12
jfdctint 3 loops	Loopfrog 1	0	3	0	5.61
	Loopfrog 2	3	0	0	0.05
	CTA	3	0	0	1.24
	Terminator	3	0	0	0.98
lms 10 loops	Loopfrog 1	3	7	0	2.86
	Loopfrog 2	6	4	0	10.49
	CTA	6	4	0	2923.12
	Terminator	6	3	1	251.03 +
ludcmp 11 loops	Loopfrog 1	0	11	0	96.73
	Loopfrog 2	5	6	0	112.81
	CTA	3	5	3	3.26 +
	Terminator	3	8	0	94.66
matmul 5 loops	Loopfrog 1	0	5	0	0.15
	Loopfrog 2	5	0	0	0.09
	CTA	3	2	0	1.97
	Terminator	3	2	0	2.15
minver 17 loops	Loopfrog 1	1	16	0	2.57
	Loopfrog 2	16	1	0	7.66
	CTA	14	1	2	105.26 +
	Terminator	14	1	2	87.09 +
qsort-exam 6 loops	Loopfrog 1	0	6	0	0.67
	Loopfrog 2	0	6	0	3.96
	CTA	0	5	1	45.92 +
	Terminator	0	5	1	2530.58 +
qurt 1 loop	Loopfrog 1	0	1	0	8.02
	Loopfrog 2	1	0	0	13.82
	CTA	1	0	0	55.65
	Terminator	0	0	1	0.00
select 4 loops	Loopfrog 1	0	4	0	0.55
	Loopfrog 2	0	4	0	3.56
	CTA	0	3	1	32.60 +
	Terminator	0	3	1	28.12 +
sqrt 1 loop	Loopfrog 1	0	1	0	0.60
	Loopfrog 2	1	0	0	5.10
	CTA	1	0	0	15.28
	Terminator	0	0	1	0.00

Table 5.3. SNU real-time benchmarks suite (continued): comparison of Loopfrog with CTA and Terminator.

Benchmark	**Method**	**T**	**NT**	**TO**	**Time**	
adpcm 11 loops	LOOPFROG 1	8	3	0	59.66	
	LOOPFROG 2	10	1	0	162.75	
	CTA	8	3	0	101.30	
	TERMINATOR	6	2	3	94.45	+
bcnt 2 loops	LOOPFROG 1	0	2	0	2.63	
	LOOPFROG 2	0	2	0	2.82	
	CTA	0	2	0	0.79	
	TERMINATOR	0	2	0	0.30	
blit 4 loops	LOOPFROG 1	0	4	0	0.16	
	LOOPFROG 2	3	1	0	0.05	
	CTA	3	1	0	5.95	
	TERMINATOR	3	1	0	3.67	
compress 18 loops	LOOPFROG 1	5	13	0	3.13	
	LOOPFROG 2	6	12	0	33.92	
	CTA	5	12	1	699.00	+
	TERMINATOR	7	10	1	474.36	+
crc 3 loops	LOOPFROG 1	1	2	0	0.15	
	LOOPFROG 2	2	1	0	0.21	
	CTA	1	1	1	0.33	+
	TERMINATOR	2	1	0	14.58	
engine 6 loops	LOOPFROG 1	0	6	0	2.40	
	LOOPFROG 2	2	4	0	9.88	
	CTA	2	4	0	16.20	
	TERMINATOR	2	4	0	4.88	
fir 9 loops	LOOPFROG 1	2	7	0	5.99	
	LOOPFROG 2	6	3	0	21.59	
	CTA	6	3	0	2957.06	
	TERMINATOR	6	2	1	193.91	+
g3fax 7 loops	LOOPFROG 1	1	6	0	1.57	
	LOOPFROG 2	1	6	0	6.05	
	CTA	1	5	1	256.90	+
	TERMINATOR	1	5	1	206.85	+
huff 11 loops	LOOPFROG 1	3	8	0	24.37	
	LOOPFROG 2	8	3	0	94.61	
	CTA	7	3	1	16.35	+
	TERMINATOR	7	4	0	52.32	
jpeg 23 loops	LOOPFROG 1	2	21	0	8.37	
	LOOPFROG 2	16	7	0	32.90	
	CTA	15	8	0	2279.13	
	TERMINATOR	15	8	0	2121.36	

Continued in the next page...

Table 5.4. Powerstone benchmark suite: comparison of LOOPFROG with CTA and TERMINATOR.

Table 5.4 – Continued

Benchmark	Method	T	NT	TO	Time	
pocsag 12 loops	LOOPFROG 1	3	9	0	2.07	
	LOOPFROG 2	9	3	0	6.91	
	CTA	9	3	0	10.39	
	TERMINATOR	7	3	2	1557.57	+
qurt 2 loops	LOOPFROG 1	0	2	0	3.56	
	LOOPFROG 2	1	1	0	11.67	
	CTA	1	1	0	30.77	
	TERMINATOR	0	0	2	0.00	
ucbqsort 15 loops	LOOPFROG 1	1	14	0	0.79	
	LOOPFROG 2	2	13	0	2.06	
	CTA	2	12	1	71.73	+
	TERMINATOR	9	5	1	51.08	+
v42 12 loops	LOOPFROG 1	0	12	0	82.84	
	LOOPFROG 2	0	12	0	2587.22	
	CTA	0	12	0	73.57	
	TERMINATOR	1	11	0	335.69	

Table 5.4. Powerstone benchmark suite (continued): comparison of LOOPFROG with CTA and TERMINATOR.

Benchmark	Method	T	NT	TO	Time	
jhead 8 loops	LOOPFROG 1	1	7	0	23.78	
	LOOPFROG 2	4	4	0	78.93	
	CTA	3	5	0	42.38	
	TERMINATOR	2	4	2	208.78	+

Table 5.5. Jhead-2.6 utility: comparison of LOOPFROG with CTA and TERMINATOR.

Benchmark	Method	T	NT	TO	Time
bchunk 9 loops	LOOPFROG 1	3	6	0	1.67
	LOOPFROG 2	3	6	0	31.16
	CTA	3	6	0	53.03
	TERMINATOR	4	5	0	91.13

Table 5.6. Bchunk 1.2.0 utility: comparison of LOOPFROG with CTA and TERMINATOR.

The aggregated data on experiments with Windows device drivers is provided in Table 5.7. The benchmarks are grouped according to the harness used

Benchmark group	Method	T	NT	TO	Time
SDV FLAT DISPATCH HARNESS 557 loops in 30 benchmarks	LOOPFROG 1	135	389	33	1752.1
	LOOPFROG 2	215	201	141	10584.4
	CTA	166	160	231	25399.5
SDV FLAT DISPATCH STARTIO HARNESS 557 loops in 30 benchmarks	LOOPFROG 1	135	389	33	1396.0
	LOOPFROG 2	215	201	141	9265.8
	CTA	166	160	231	28033.3
SDV FLAT HARNESS 635 loops in 45 benchmarks	LOOPFROG 1	170	416	49	1323.0
	LOOPFROG 2	239	205	191	6816.4
	CTA	201	186	248	31003.2
SDV FLAT SIMPLE HARNESS 573 loops in 31 benchmarks	LOOPFROG 1	135	398	40	1510.0
	LOOPFROG 2	200	191	182	6814.0
	CTA	166	169	238	30292.7
SDV HARNESS DRIVER CREATE 9 loops in 5 benchmarks	LOOPFROG 1	1	8	0	0.1
	LOOPFROG 2	1	8	0	0.2
	CTA	1	8	0	151.8
SDV HARNESS PNP DEFERRED IO REQUESTS 177 loops in 31 benchmarks	LOOPFROG 1	22	98	57	47.9
	LOOPFROG 2	66	54	57	617.4
	CTA	80	94	3	44645.0
SDV HARNESS PNP IO REQUESTS 173 loops in 31 benchmarks	LOOPFROG 1	25	94	54	46.6
	LOOPFROG 2	68	51	54	568.7
	CTA	85	86	2	15673.9
SDV PNP HARNESS SMALL 618 loops in 44 benchmarks	LOOPFROG 1	172	417	29	8209.5
	LOOPFROG 2	261	231	126	12373.2
	CTA	200	177	241	26613.7
SDV PNP HARNESS 635 loops in 45 benchmarks	LOOPFROG 1	173	426	36	7402.2
	LOOPFROG 2	261	230	144	13500.2
	CTA	201	186	248	41566.6
SDV PNP HARNESS UNLOAD 506 loops in 41 benchmarks	LOOPFROG 1	128	355	23	8082.5
	LOOPFROG 2	189	188	129	13584.6
	CTA	137	130	239	20967.8
SDV WDF FLAT SIMPLE HARNESS 172 loops in 18 benchmarks	LOOPFROG 1	27	125	20	30.3
	LOOPFROG 2	61	91	20	202.0
	CTA	73	95	4	70663.0

Table 5.7. Aggregated data of the comparison between LOOPFROG and CTA on Windows device drivers

upon extraction of a model with GOTO-CC. Note that we skip the benchmarks where no loops are detected. Therefore, groups in Table 5.7 have different number of benchmarks/loops to report. Also, we do not report TERMINATOR results here because, as it was shown in [KSTW10], CTA outperforms it on these big benchmarks.

Discussion of experiments

Note, that direct comparison of LOOPFROG in time with iterative techniques like CTA and TERMINATOR is not fair. The latter methods are complete at least for finite-state programs, relative to the completeness of ranking synthesis method (which is not complete by default in the current CTA/TERMINATOR implementation for scalability reasons). Our loop summarization technique on the other hand is a static analysis which aims only for conservative abstractions. In particular, it does not try to prove unreachability of a loop or of preconditions that lead to non-termination.

The timing information provided here serves as a reference that allows to compare efforts of achieving the same result. Note that:

- LOOPFROG spends time enumerating invariant candidates, provided by the chosen abstract domain, against a path of one loop iteration. Compositionality and d.wf. checks are not required for the chosen domains.

- CTA spends time 1) unwinding loop iterations, 2) discovering a ranking function for each unwounded path and 3) checking compositionality of a discovered relation.

- TERMINATOR spends time 1) enumerating all paths through the loop and 2) discovering a ranking function for each path.

The techniques can greatly vary in time of dealing with a particular loop/program. CTA and TERMINATOR give up on a loop once a they hit a path on which ranking synthesis fails. LOOPFROG gives up on a loop if it runs out of transition invariant candidates to try. In a few tests it leads to an advantage of TERMINATOR (`huff` and `engine` in Table 5.4), however, we observe in almost all other tests that the LOOPFROG technique is generally cheaper (often in orders of magnitude) in computational efforts required for valid termination argument discovery.

Tables 5.3, 5.4 and 5.5 show that loop summarization is able to prove termination for the same number of loops as CTA and TERMINATOR, but does so with less resource requirements. In particular it demonstrates that a simple strict order relation for all numeric variables of the loop (Table 5.1, domain #1) is, in practice, as effective as CTA with default ranking functions. The results on the considerably larger Windows device drivers (Table 5.7) lead to similar conclusions.

The comparison demonstrates some weak points of the iterative analysis:

- Enumeration of all paths through the loop can require many iterations or even can be infinite for infinite state systems (as are most of realistic programs).

- The ranking procedures can often fail to produce a ranking argument; but if it succeeds, a simple relation has could be enough as well.

- The search for a compositional transition invariant sometimes results in exponential growth of required loop unrollings (in case of CTA).

LOOPFROG does not suffer from at least the first of these problems: the analysis of each loop requires a finite number of calls to a decision procedure. The second issue is leveraged by relative simplicity of adding new abstract domain over implementing complex ranking function method. The third issue is transformed into generation of suitable invariant candidates, which, in general, may generate many candidates, i.e., that also slows the procedure down. However we can control the order of candidates by prioritizing some domains over the others, thus, can expect simple ranking arguments to be discovered earlier.

The complete results of experiments as well as the LOOPFROG tool, are available at `www.verify.inf.usi.ch/loopfrog/termination`.

5.6 Related work and summary

Although the field of program termination analysis is relatively old and the first results date back to Turing [Tur49], recent years have seen a tremendous increase in practical applications of termination proving. Two directions of research enabled the efficacy of termination provers in practice:

- Transition invariants by Podelski and Rybalchenko [PR04b], and

- the size-change termination principle (SCT) by Lee, Jones and Ben-Amram [LJBA01],

where the latter has its roots in previous research on termination of declarative programs. Until very recently, these two lines of research did not intersect much. The first systematic attempt to understand their common treats is a very recent publication by Heizmann et al. [HJP11].

5.6.1 Relation to size-change termination principle

Termination analysis based on the SCT principle usually involves two steps:

1. construction of an abstract model of the original program in the form of *size-change graphs* (SC-graphs) and

2. analysis of the SC-graphs for termination.

SC-graphs contain abstract program values as nodes and use two types of edges, along which values of variables *must decrease*, or *decrease or stay the same*. No edge between nodes means that none of the relations can be ensured. Graphs G which are closed under composition with itself, are called *idempotent*, i.e., $G; G = G$.[4]

Lee et al. [LJBA01] identify two termination criteria based on a size-change graph:

1. The SC-graph is well-founded, or

2. The idempotent components of an SC-graph are well-founded.

An SC-graph can be related to transition invariants as follows. Each subgraph corresponds to a conjunction of relations, which constitutes a transition invariant. The whole graph forms a disjunction, resulting in a termination criterion very similar to that presented as Theorem 1: if an SC-graph is well-founded then there exists a d.wf. transition invariant. Indeed, Heizmann et al. [HJP11] identify the SCT termination criterion as strictly stronger than the argument via transition invariants [PR04b]. In other words, there are terminating programs for which there are no suitable SC-graphs that comply with termination criteria above.

The intuition behind SCT termination being a stronger property comes from the fact that SC-graphs abstract from the reachability of states in a program, i.e., SC-graph requires termination of all paths regardless of whether those paths are reachable or not. Transition invariants, on the other hand, require the computation of the reachable states of the program. In this respect our lightweight analysis is closely related to SCT, as it havocs the input to individual loop iterations before checking a candidate transition invariant.

The domains of SC-graphs correspond to abstract domains in our approach. The initial inspiration for the domains we experimented with comes from a

[4]In this discussion we omit introducing the notation necessary for a formal description of SCT; see Lee et al. [LJBA01, HJP11] for more detail.

recent survey on ranking functions for SCT [BAL09]. The domains #1-4 in Table 5.1 encode those graphs with only down-arcs. Domain #5 has down-arcs and edges that preserve the value. However, note that, in order to avoid well-foundedness checks, we omit domains that have mixed edge types.

Program abstraction using our loop summarization algorithm can be seen as construction of size-change graphs. The domains suggested in Section 5.4 result in SC-graphs that are idempotent and well-founded by construction.

Another relation to SCT is the second SCT criterion based on idempotent SC-components. In [HJP11] the relation of idempotency to some notion in transition invariant-based termination analysis was stated as an open question. However, there is a close relation between the idempotent SC-components and compositional transition invariants (Definition 18, page 90) used here and in compositional termination analysis [KSTW10]. The d.wf. transition invariant constructed from idempotent graphs is also a compositional transition invariant.

5.6.2 Relation to other research in transition invariant-based termination

The work in this Chapter is a continuation of the research of transition invariants-based termination proving methods initiated by [PR04b]. Methods developed on the basis of transition invariants rely on an iterative abstraction refinement-like construction of d.wf. transition invariants [CPR05, CPR06, KSTW10]. Our approach is different, because it aims to construct a d.wf. transition invariant without refinement. Instead of ranking function discovery for every non-ranked path, we use abstract domains that express ranking arguments for all paths at the same time.

Chawdhary et al. [CCG$^+$08] propose a termination analysis using a combination of fixpoint-based abstract interpretation and an abstract domain of disjunctively well-founded relations. The abstract domain they suggest is of the same form as domain #5 in Table 5.1. However their technique attempts iterative computation of the set of abstract values and has a fixpoint detection of the form $T \subseteq R^+$, while in our approach it is enough to check $T \subseteq R$, combined with the compositionality criterion. This allows more abstract domains to be applied for summarization, as each check is less demanding on the theorem prover.

Dams et al. [DGG00] present a set of heuristics that allow heuristic inference of candidate ranking relations from a program. These heuristics can be seen as abstract domains in our framework. Moreover, we also show how candidate relations can be checked effectively.

Cook et al. [CPR09] use relational predicates to extend the framework of Reps et al.[RHS95] to support termination properties during computation of inter-procedural program summaries. Our approach shares a similar motivation and adds termination support to abstract domain-based loop summarization. However, we concentrate on scalable non-iterating methods to construct the summary while Cook et al. [CPR09] rely on a refinement-based approach. The same argument applies in the case of Balaban et al.'s framework[BCP06] for procedure summarization with liveness properties support.

Berdine et al. [BCC^{+}07] use the Octagon and Polyhedra abstract domains to discover invariance constraints sufficient to ensure termination. Well-foundedness checks, which we identify as an expensive part of the analysis, are left to iterative verification by an external procedure like in the TERMINATOR algorithm [CPR06] and CTA [KSTW10]. In contrast to these methods, our approach relies on abstract domains which are well-founded by construction and therefore do not require explicit checks.

Dafny, a language and a program verifier for functional correctness [Lei10], employs a very similar method to prove a loop (or a recursive call) termination. First, each Dafny type has an ordering, values are finite, and, except for integers, values are bounded from below. Second, Dafny offers a special `decrease` predicate. Now, if one can provide a tuple of variables (a termination metric) for which `decrease` holds then termination can be concluded (note below a special case of integer variables). A termination metric can be suggested by a developer or guessed using predefined heuristics. Effectively, this method maps one to one to strict order relational domains used in LOOPFROG.

It is interesting to note one particular case in Dafny, that is, if integer variable is used in termination metric than an additional invariant is required, namely, the existence of the minimal element in ordering. LOOPFROG usually deals with machine integers and, thus, enjoys having a minimum by design. But in Dafny integers are unbounded from below and, therefore, it has to prove a simple invariant for integer, for instance, check if the integer variable is ≥ 0. Thus, Dafny already combines state and transition invariants, the research directions which is yet to be explored in LOOPFROG.

Altogether, successful usage of the relatively simple predefined heuristics in Dafny and our experiments supports the main claim of this chapter: lightweight analysis based on simple heuristics is often enough to prove many loops terminating.

Summary

In this Chapter we present an extension to a loop summarization algorithm such that it correctly handles termination properties while constructing a loop-less program over-approximation. To that end, we employ abstract domains that encode transition invariants, i.e., relations over pre- and post-state of the summarized loop. Termination of loops may be established at the same time, by checking compositionality and disjunctive well-foundedness of the discovered transition invariants. We implemented the algorithm in the LOOPFROG tool (`www.verify.inf.usi.ch/loopfrog`) and demonstrate the practicality of our approach on a large set of benchmarks including open-source programs and Windows device drivers. Though theoretically weaker than existing path exploration-based algorithms like TERMINATOR, it achieves the comparable practical result with a solid advantage in time performance. Besides, the method suggests that existing abstract domain-based abstraction techniques can be extended to support reasoning about termination. For that, compositionality and d.wf.-ness should be established either for the individual invariants or for the abstract domains used to discover them.

Chapter 6

Conclusion

If you know where you're going
— you aren't lost!
Sport orienteering wisdom

The discovery of an appropriate abstraction is a fundamental step in establishing a successful verification framework. Abstraction not only reduces the computational burden of verification, but also makes it possible to verify infinite-state software models. Also, as noted by Clarke, Grumberg and Peled in the seminal "Model checking" book [CGP99]:

> *Abstraction is probably the most important technique for reducing the state explosion problem.*

This thesis is a step towards understanding of what is a right abstraction, how it should be discovered and used to enable efficient analysis of programs by formal verification tools.

6.1 Contributions

Synergy of fast and precise abstraction in the abstraction-refinement loop First, this thesis reports on a new approach to the abstraction refinement that combines precise and approximated techniques. The new algorithm benefits from the precise abstraction computation, as it allows to avoid too many iterations due to spurious transitions of the abstract model. At the same time, fast abstraction computation helps to discover more of previously untouched spurious behaviors. Moreover, by exploiting the localized-abstraction framework, algorithm reduces the abstraction computation to the parts of the system that are relevant to the property and keeps the approximated abstraction in all parts of the program that are irrelevant to prove the property. This technique is orthogonal to any particular abstraction or refinement procedure and can be used for any existing abstraction-refinement combination.

The conducted evaluation compares the new technique with the classical precise and imprecise algorithms. These tests with various benchmarks show that the new approach systematically outperforms both precise and imprecise techniques. The results confirm that the new technique achieves the goal of reducing the number of iterations of the CEGAR loop.

Based on evaluation results we also developed a threshold-based optimization that further restricts precise computation in order to avoid unnecessary application of expensive quantifier elimination.

Program abstraction by loop summarization Second, a novel algorithm for program abstraction using symbolic abstract transformers is described. The algorithm computes an abstract model of a program with respect to a given abstract interpretation by replacing loops and function calls in the control flow graph by their symbolic transformers. The run-time of the new algorithm is linear in the number of looping constructs in a program and a finite number of (relatively simple) decision procedure calls is used for discovery of every abstract symbolic transformer. Therefore, it addresses the perennial problem of the high complexity of computing abstract fixpoints.

The procedure over-approximates the original program, which implies soundness of the analysis, but, as any other abstraction-based technique, it can introduce false positives on the consequent phase of analysis of the constructed abstract model. An additional benefit of the technique is its ability to generate leaping counterexamples, which are helpful for diagnosis of the error or for filtering spurious warnings. The conducted experimental evaluation demonstrates the best error-detection and error-discrimination rates comparing to a broad selection of static analysis tools.

Light-weight program termination analysis Third, this thesis describes an approach for the light-weight program termination analysis. For a sequential program, termination of all loops is enough to conclude program termination, therefore we focused our analysis on individual loops. The new algorithm is based on loop summarization and employs relational abstract domains to discover transition invariants for loops. It uses compositionality of a transition invariant as a completeness criterion, i.e., that a discovered transition invariant holds for any execution through the loop. If such an invariant exists and it is (disjunctively) well-founded, then the loop is guaranteed to terminate. Well-foundedness can be checked either by an application of a quantifier-supporting decision procedure or be ensured by construction. In the latter case an abstract domain for producing candidates for transition invariants should be chosen appropriately.

Note, that, although this algorithm is theoretically incomplete (because a compositional transition invariant does not always exist and the ability to discover transition invariants is restricted by expressiveness of the selected abstract domains) the practical evaluation demonstrates its effectiveness. We applied new termination analysis to numerous benchmarks including Windows device drivers and demonstrated high scalability as well as precision level that matches to the state-of-the-art path enumeration-based algorithms such as TERMINATOR and CTA.

In contrast to other methods, our algorithm performs both loop summarization and transition invariant inference at the same time, thus, both safety- and liveness properties of loop semantics are preserved. Also, it utilizes a family of simple, custom abstract domains whereas other works in termination analysis often use off-the-shelf domains; it seems very interesting to note that simpler domains can go a long way in solving those problems, while keeping computational costs low.

Implementations The algorithm that combines fast and precise abstraction was implemented on top of a CEGAR-based model checker SATABS. The program summarization algorithm and program termination analysis were implemented in our tool LOOPFROG, that performs static analysis for ANSI-C programs. The tools and the experimental results (reported in this thesis) are available respectively at:

- `www.verify.inf.usi.ch/projects/synergy`
- `www.verify.inf.usi.ch/loopfrog`
- `www.verify.inf.usi.ch/loopfrog/termination`

6.2 Future Work

Loop summarization as a helper to loop invariant-based code motion Loop invariant-based code motion is known to be an effective code optimization during compilation [ASU86]. Recently it received an additional attention because of active research in just-in-time compilation for dynamic typed languages such as JavaScript and PHP. Despite it is complicated by the dynamic nature of the targeted languages, we believe specially designed abstract domains for both state and transition invariants can help to discover assumptions (e.g., loop termination, variable bounds, branch frequencies) that compiler can rely on for the code motion.

Loop invariants to speed up paths exploration for symbolic execution Engineering advances in symbolic execution for software testing purposes (e.g. DART [GKS05], SAGE [GLM08], KLEE [CDE08], S2E [CKC11]) stressed out the problem of *paths explosion* in program analysis — a problem of a rapid increase in a number of paths that have to be considered for a thorough testing of a software. Paths explosion is a particular form of state-space explosion, and, the same way as a latter one, is often caused by loops in a program. To fight it one may apply problem-tailored abstraction of several paths in one satisfying a particular criteria. If a technique can also be effectively deployed within an existing symbolic execution framework the speed up in paths exploration can be achieved.

We consider loop summarization as a potential candidate to address the paths explosion problem since it focuses on abstraction of loop behaviors. As a future work we would like to identify abstract domains that can be used for paths grouping and evaluate it within a path exploration framework or an automated testing framework.

Problem-specific abstract domains for termination (liveness) analysis Additional relational abstract domains should be considered for termination (liveness) analysis in areas where it is appropriate. Possible applications include:

- Verification of liveness properties in protocols implementations with abstract domains used to ensure the message ordering.
- Verification of liveness properties in concurrent programs with abstract domains employed to express the independence of termination from thread scheduling or the execution progress over all threads at once.

Recent attempt to build a bridge between transition invariants-based termination analysis and size-change termination by Heizmann et al. [HJP11] suggests that relatively well-studied size-change graphs can be adopted as abstract domains.

As a stand-alone theoretical program we see definition of class of systems and properties for which termination (liveness) can be proved by induction, i.e., by guess and proof of the base step (discovery of a transition invariant) and proof of the inductive step (compositionality of a transition invariant).

Loop invariants on demand for dynamic analysis Dynamic analysis (i.e., analysis of program traces) is a widely applied instrument in software engineering. It can be used, for instance, to discover *likely* program invariants [EPG$^+$07] that can be valuable for compilation optimization, testing, thread scheduling, etc. Instead of likely invariants we want to contribute with real program invariants (both state and transition ones), but discover them on demand, based on abstract domains defined using the constraints from dynamic analysis.

Combined state/transition invariants abstract domains for conditional termination (liveness) Cook et. al. aim to compute a loop precondition that implies termination and use it to establish *conditional termination* [CGLA$^+$08]. A combined state/transition invariants can be used for the similar purpose.

It is also interesting to figure out if the negative result of transition invariant candidate check can be mapped directly to a non-terminating counterexample or precondition. For instance, the failure to ensure the total order between all values of iterator i in a loop may hint the integer overflow that leads to non-termination.

Integration of the fast/precise abstraction within other abstraction-refinement loops (e.g., interpolation-based model checking) It is interesting to implement the combined fast/precise abstraction approach in tools that are based on interpolation for predicate discovery [HJMM04, JM06]. Another direction to consider is the investigation of the same trade-off between precise and approximated approaches in the context of purely interpolation-based model checking[McM06], which does not need predicate abstraction.

Besides, it may be interesting to establish fine-grained correspondence between the semantics of the analyzed model (e.g., semantics of C code instructions), predicate discovery and the combination of fast/precise abstraction.

Appendix A

Additional experimental data

In Chapter 3 we reported an experimental evaluation of abstraction-refinement techniques on the Verisec test suite by Ku et al. [KHCL07]. Table A.1 reports the raw results of the experiment, which were used to draw plots in Figures 3.3, 3.4, 3.5, 3.6, 3.7 and 3.8 in Section 3.4.2.

Figure A.1 presents an example of a class that implement an abstract domain in LOOPFROG (as presented in Chapter 4).

	WP			NewST			NewST with $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
apache-cve-2004-0940-1	7	38.382	67	5	26.041	67	5	26.207	67	4	666.526	67	5	47.657	67	5	48.32	67
apache-cve-2004-0940-1-1	6	31.897	65	6	34.881	65	6	34.729	65	4	313.053	65	6	55.484	65	5	47.833	65
apache-cve-2004-0940-1-1-fixed	5	26.931	66	5	28.732	66	5	28.802	66	3	478.553	66	4	54.69	66	4	57.397	66
apache-cve-2004-0940-1-2	7	37.525	67	5	27.754	67	5	28.191	67	4	668.031	67	5	48.762	67	5	48.229	67
apache-cve-2004-0940-1-2-fixed	7	43.26	68	5	32.067	68	5	32.116	68	4	667.251	68	6	59.741	68	5	51.355	68
apache-cve-2004-0940-1-fixed	7	43.492	68	5	31.005	68	5	30.926	68	4	682.436	68	5	56.138	68	5	57.956	68
apache-cve-2004-0940-2	7	51.462	71	5	36.866	71	5	37.297	71	4	363.886	71	6	67.817	71	5	56.826	71
apache-cve-2004-0940-2-1	6	36.272	68	5	27.425	68	5	27.943	68	4	192.519	68	5	52.237	68	5	53.954	68
apache-cve-2004-0940-2-1-fixed	5	35.784	70	5	37.491	70	5	37.558	70	3	289.3	70	4	84.876	70	4	85.206	70
apache-cve-2004-0940-2-2	7	53.668	71	5	36.706	71	5	36.619	71	4	362.152	71	5	63.484	71	5	57.456	71
apache-cve-2004-0940-2-2-fixed	7	54.446	72	5	36.566	72	5	36.481	72	4	372.369	72	6	79.064	72	5	60.049	72
apache-cve-2004-0940-2-fixed	7	55.393	72	5	38.478	72	5	38.141	72	4	372.055	72	5	65.181	72	5	66.65	72
apache-cve-2004-0940-3	9	13.68	55	10	17.035	55	10	17.293	55	5	388.281	55	6	16.01	55	6	16.737	55
apache-cve-2004-0940-3-1	10	14.802	50	8	14.095	50	8	14.28	50	5	149.978	50	7	18.755	50	7	18.356	50
apache-cve-2004-0940-3-1-fixed	7	12.973	54	6	15.046	54	6	15.277	54	4	344.418	54	5	16.595	54	5	16.892	54
apache-cve-2004-0940-3-2	9	13.411	55	9	15.875	55	9	15.594	55	5	389.672	55	6	17.702	55	6	17.879	55
apache-cve-2004-0940-3-2-fixed	10	16.587	56	10	17.179	56	10	17.241	56	5	421.188	56	6	16.078	56	6	16.089	56
apache-cve-2004-0940-3-fixed	9	17.772	56	9	18.206	56	9	18.275	56	5	394.509	56	6	19.319	56	6	19.678	56
apache-cve-2004-0940-4	10	18.094	59	10	17.954	59	10	17.666	59	5	271.291	59	6	17.566	59	6	17.167	59
apache-cve-2004-0940-4-1	10	14.415	53	8	11.183	53	8	11.21	53	5	94.494	53	7	16.296	53	7	15.907	53
apache-cve-2004-0940-4-1-fixed	7	15.426	58	6	13.27	58	6	13.181	58	4	256.317	58	6	25.8	58	5	22.67	58
apache-cve-2004-0940-4-2	10	18.677	59	10	18.315	59	10	18.572	59	5	271.209	59	6	17.204	59	6	17.447	59
apache-cve-2004-0940-4-2-fixed	10	19.185	60	10	18.805	60	10	18.809	60	5	299.217	60	6	17.393	60	6	17.454	60
apache-cve-2004-0940-4-fixed	10	19.928	60	9	17.624	60	9	17.914	60	5	299.249	60	5	15.789	60	5	15.901	60
apache-cve-2004-0940-5	8	43.835	68	6	34.537	68	6	34.44	68	4	678.552	68	5	49.571	68	5	50.345	68
apache-cve-2004-0940-5-2	7	42.868	66	5	31.092	66	5	29.475	66	4	328.418	66	6	58.847	66	5	51.036	66
apache-cve-2004-0940-5-3	7	39.585	68	5	29.733	68	5	29.848	68	4	680.786	68	5	50.161	68	5	52.143	68
apache-cve-2004-0940-6	14	217.291	104	10	173.377	104	10	169.442	104	5	1333.36	104	7	138.341	104	8	151.159	104
apache-cve-2004-0940-6-2	11	146.932	96	12	194.611	96	12	204.809	96	5	899.996	96	7	112.496	96	6	105.604	96
apache-cve-2004-0940-6-3	10	168.331	104	9	159.87	104	9	162.528	104	5	1328.03	104	7	131.381	104	8	146.018	104
apache-cve-2004-0940-7	10	17.793	56	9	16.207	56	9	16.227	56	5	411.847	56	6	17.005	56	6	18.032	56
apache-cve-2004-0940-7-2	9	13.951	51	8	15.416	51	8	15.284	51	5	159.7	51	7	19.36	51	7	19.675	51
apache-cve-2004-0940-7-3	9	17.321	56	9	17.282	56	9	17.288	56	5	412.389	56	6	18.137	56	6	18.16	56
apache-cve-2004-0940-8	28	306.202	100	16	153.517	100	16	143.309	100	7	1910.87	100	16	234.382	100	11	141.013	100
apache-cve-2004-0940-8-2	26	168.444	84	17	99.332	84	17	100.656	84	7	807.101	84	11	92.093	84	10	74.358	84
apache-cve-2004-0940-8-3	26	292.398	100	17	168.996	100	19	187.085	100	7	1918.83	100	14	186.216	100	11	132.156	100
apache-cve-2006-3747-1-fixed	3	0.692	5	3	0.681	5	3	0.689	5	2	0.345	5	2	0.353	5	2	0.35	5
apache-cve-2006-3747-2-fixed	3	0.628	5	3	0.621	5	3	0.629	5	2	0.332	5	2	0.338	5	2	0.331	5
apache-cve-2006-3747-3	3	0.436	10	3	0.424	10	3	0.45	10	3	0.664	10	3	0.61	10	3	0.576	10

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k=7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
apache-cve-2006-3747-3-fixed	3	0.326	5	3	0.33	5	3	0.314	5	2	0.131	5	2	0.14	5	2	0.137	5
apache-cve-2006-3747-4	23	627.591	177	19	615.106	177	19	608.128	177		—			—			—	
apache-cve-2006-3747-4-fixed	3	0.388	5	3	0.386	5	3	0.389	5	2	0.166	5	2	0.183	5	2	0.176	5
apache-cve-2006-3747-5	37	2539.81	177	27	1605.53	177	26	1587.83	177		—			—			—	
apache-cve-2006-3747-5-fixed	3	0.491	5	3	0.493	5	3	0.492	5	2	0.235	5	2	0.242	5	2	0.244	5
apache-cve-2006-3747-6	20	197.009	104	18	177.264	104	18	178.353	104		—			—			—	
apache-cve-2006-3747-6-fixed	3	0.484	5	3	0.446	5	3	0.432	5	2	0.199	5	2	0.216	5	2	0.207	5
bind-cve-2001-0011-1	12	49.471	65	10	46.087	65	10	45.592	65		—		10	131.789	65	9	124.332	65
bind-cve-2001-0011-1-fixed	2	1.784	4	2	1.756	4	2	1.762	4	2	1.79	4	2	1.817	4	2	1.793	4
bind-cve-2001-0011-2	11	18.745	45	9	15.629	45	9	15.52	45		—		9	48.47	45	8	48.628	45
bind-cve-2001-0011-3	5	3.045	34	5	3.373	34	5	3.41	34	4	547.341	34	4	26.192	34	4	26.39	34
bind-cve-2001-0011-3-fixed	2	0.388	4	2	0.391	4	2	0.38	4	2	0.396	4	2	0.392	4	2	0.412	4
edbrowse-cve-2006-6909-1	15	84.19	101	14	84.657	101	14	84.487	101		—		14	310.615	101	11	217.538	101
edbrowse-cve-2006-6909-1-fixed	2	0.249	4	2	0.254	4	2	0.25	4	2	0.256	4	2	0.293	4	2	0.265	4
edbrowse-cve-2006-6909-2	8	9.105	55	8	9.506	55	8	9.425	55		—		7	192.605	55	7	195.318	55
edbrowse-cve-2006-6909-2-fixed	2	0.167	4	2	0.172	4	2	0.173	4	2	0.178	4	2	0.188	4	2	0.184	4
edbrowse-cve-2006-6909-3-fixed	2	0.308	4	2	0.302	4	2	0.305	4	2	0.32	4	2	0.326	4	2	0.333	4
gxine-cve-2007-0406	3	0.285	11	3	0.282	11	3	0.279	11	3	1.19	11		—			—	
gxine-cve-2007-0406-fixed	2	0.106	4	2	0.105	4	2	0.105	4	2	0.11	4	2	0.119	4	2	0.117	4
libgd-cve-2007-0455-1	7	14.339	48	7	14.738	48	7	14.61	48		—		17	68.406	45	9	31.288	45
libgd-cve-2007-0455-2	21	47.568	70	14	35.545	70	14	35.823	70		—		18	724.463	70	15	721.551	70
libgd-cve-2007-0455-2-fixed	33	272.776	86	29	253.244	86	29	263.289	86		—		28	1332.62	81	23	1369.32	86
libgd-cve-2007-0455-3	9	4.32	32	10	5.527	32	10	5.466	32	5	129.556	32	9	7.148	32	8	6.561	32
libgd-cve-2007-0455-3-fixed	25	31.874	53	19	35.98	53	20	37.662	53		—		16	102.48	66	16	192.869	66
libgd-cve-2007-0455-4	10	11.177	40	8	9.228	40	8	9.342	40	5	247.666	40	12	23.092	40	7	11.722	40
libgd-cve-2007-0455-4-fixed	57	1594.26	124	35	2902.68	124	36	1325.79	124		—			—			—	
madwifi-cve-2006-6332-1-1	4	1.354	28	4	1.408	28	4	1.379	28	3	41.925	28	3	3.362	28	3	3.427	28
madwifi-cve-2006-6332-1-1-fixed	2	0.408	20	2	0.409	20	2	0.42	20	2	2.597	20	2	2.472	20	2	2.487	20
madwifi-cve-2006-6332-1-fixed	2	0.387	20	2	0.384	20	2	0.39	20	2	2.34	20	2	2.382	20	2	2.326	20
madwifi-cve-2006-6332-2	5	2.158	33	5	2.087	33	5	2.026	33	4	89.997	33		—			—	
madwifi-cve-2006-6332-2-1-fixed	3	1.052	27	3	1.08	27	3	1.045	27	3	13.65	27		—			—	
madwifi-cve-2006-6332-2-fixed	3	0.993	27	3	0.969	27	3	0.996	27	3	12.666	27		—			—	
madwifi-cve-2006-6332-3	1	0.055	2	1	0.055	2	1	0.055	2	1	0.06	2	1	0.061	2	1	0.06	2
madwifi-cve-2006-6332-3-fixed	2	0.141	6	2	0.141	6	2	0.141	6	2	0.161	6	2	0.174	6	2	0.169	6
netbsd-libc-cve-2006-6652-10-fixed	4	3.995	36	4	4.094	36	4	4.009	36	4	174.489	36	4	7.158	36	4	7.133	36
netbsd-libc-cve-2006-6652-11	3	0.787	28	3	0.793	28	3	0.805	28	3	532.27	28	3	29.98	28	3	30.091	28
netbsd-libc-cve-2006-6652-12	3	0.715	29	3	0.702	29	3	0.706	29	3	19.491	29	3	2.074	29	3	2.041	29
netbsd-libc-cve-2006-6652-1-fixed	2	0.054	2	2	0.054	2	2	0.053	2	1	0.028	2	1	0.028	2	1	0.026	2

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for `WP`, `NewST`, `NewST` with a threshold $k = 7$, `SATQE`, `NewSP` and `NewST` +`NewSP`. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

Benchmark	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
	Iterations	Time	Predicates	Iterations	Time	Predicates	Iterations	Time	Predicates	Iterations	Time	Predicates	Iterations	Time	Predicates	Iterations	Time	Predicates
netbsd-libc-cve-2006-6652-2	5	22.43	76	5	21.679	76	5	21.28	76		—		4	1498.19	76	4	1498.85	76
netbsd-libc-cve-2006-6652-2-fixed	6	37.709	76	6	39.959	76	6	37.127	76		—		4	1505.19	76	4	1497.82	76
netbsd-libc-cve-2006-6652-3	5	23.9	78	5	24.321	78	5	24.376	78		—		4	28.701	78	4	28.789	78
netbsd-libc-cve-2006-6652-3-fixed	6	39.805	78	6	37.55	78	6	36.651	78		—		4	28.241	78	4	27.971	78
netbsd-libc-cve-2006-6652-4-1	5	4.708	43	5	4.761	43	5	4.72	43		—		5	24.313	43	5	24.112	43
netbsd-libc-cve-2006-6652-4-1-fixed	9	31.15	65	9	30.698	65	9	31.365	65		—		7	68.657	65	7	67.895	65
netbsd-libc-cve-2006-6652-4-2	5	37.179	76	5	30.6	76	5	30.53	76		—		4	1522.66	76	4	1533.75	76
netbsd-libc-cve-2006-6652-4-2-fixed	6	56.839	76	6	54.659	76	6	50.421	76		—		4	1514.69	76	4	1518.51	76
netbsd-libc-cve-2006-6652-4-3	5	11.748	57	5	13.532	57	5	13.738	57		—		4	66.807	57	4	67.455	57
netbsd-libc-cve-2006-6652-4-3-fixed	6	18.985	57	6	24.316	57	6	20.735	57		—		4	66.201	57	4	66.379	57
netbsd-libc-cve-2006-6652-4-fixed	1	0.077	2	1	0.076	2	1	0.077	2	1	0.087	2	1	0.085	2	1	0.086	2
netbsd-libc-cve-2006-6652-5	5	6.403	44	5	6.481	44	5	6.432	44	4	62.531	44	5	9.583	44	5	9.513	44
netbsd-libc-cve-2006-6652-5-1	5	39.865	78	5	41.703	78	5	41.803	78		—		4	60.47	78	4	57.728	78
netbsd-libc-cve-2006-6652-5-1-fixed	6	65.527	78	6	72.324	78	6	71.646	78		—		4	45.81	78	4	45.68	78
netbsd-libc-cve-2006-6652-5-2	5	13.077	59	5	13.246	59	5	13.169	59		—		4	18.102	59	4	14.896	59
netbsd-libc-cve-2006-6652-5-2-fixed	6	20.889	59	6	21.099	59	6	21.143	59		—		4	17.636	59	4	14.698	59
netbsd-libc-cve-2006-6652-5-fixed	8	33.132	66	8	35.534	66	8	36.095	66		—		7	26.185	66	7	26.369	66
netbsd-libc-cve-2006-6652-6	5	0.834	25	5	0.799	25	5	0.788	25	4	48.978	25	4	1.53	25	4	1.551	25
netbsd-libc-cve-2006-6652-6-fixed	8	3.604	42	8	3.432	42	8	3.378	42	5	534.367	42	5	3.187	42	5	3.198	42
netbsd-libc-cve-2006-6652-7	5	9.689	64	5	9.447	64	5	9.373	64		—		4	976.765	64	4	979.149	64
netbsd-libc-cve-2006-6652-7-fixed	6	17.718	64	6	17.634	64	6	17.608	64		—		4	972.814	64	4	977.011	64
netbsd-libc-cve-2006-6652-8	5	10.74	66	5	10.731	66	5	10.683	66		—		4	16.296	66	4	16.306	66
netbsd-libc-cve-2006-6652-8-fixed	6	18.852	66	6	18.805	66	6	18.813	66		—		4	14.233	66	4	14.374	66
netbsd-libc-cve-2006-6652-9	3	2.182	35	3	2.15	35	3	2.218	35	3	545.713	35	3	36.401	35	3	36.622	35
netbsd-libc-cve-2006-6652-9-1-fixed	2	0.654	22	2	0.647	22	2	0.639	22	2	2.156	22	2	2.15	22	2	2.124	22
netbsd-libc-cve-2006-6652-9-2-fixed	2	0.652	22	2	0.646	22	2	0.788	22	2	2.158	22	2	2.133	22	2	2.151	22
openser-cve-2006-6749-10-fixed	4	8.232	26	3	23.585	26	3	23.368	26	2	101.056	26	3	102.625	26	3	117.428	26
openser-cve-2006-6749-11	19	13.582	41	8	29.658	41	8	29.438	41	3	429.893	41	4	60.243	41	4	74.926	41
openser-cve-2006-6749-1-1-fixed	4	16.334	38	4	34.576	38	4	34.546	38	2	846.763	38	3	1243.57	38	3	1250.72	38
openser-cve-2006-6749-11-fixed	6	8.861	23	3	23.66	23	3	23.631	23	2	33.959	23	3	30.595	23	3	45.781	23
openser-cve-2006-6749-12-fixed	4	7.46	23	3	23.845	23	3	23.731	23	2	33.722	23	3	30.744	23	3	45.924	23
openser-cve-2006-6749-13	200	1987.89	52	26	146.573	85	25	173.931	85		—		19	2272.4	70	13	2445.45	70
openser-cve-2006-6749-13-fixed	10	16.863	30	5	29.688	30	5	29.777	30	2	144.63	30	3	147.263	30	3	162.97	30
openser-cve-2006-6749-14-fixed	4	9.5	30	4	27.888	30	4	28.032	30	2	144.458	30	3	147.763	30	3	162.319	30
openser-cve-2006-6749-15	69	143.483	46	10	44.857	46	11	44.357	46	3	2136.99	46	7	96.008	46	5	105.747	46
openser-cve-2006-6749-15-fixed	10	14.533	29	5	26.831	29	5	26.72	29	2	62.006	29	3	60.898	29	3	75.898	29
openser-cve-2006-6749-16-fixed	4	8.463	29	4	25.384	29	4	25.344	29	2	61.779	29	3	61.023	29	3	76.148	29
openser-cve-2006-6749-17	77	317.923	44	13	52.11	44	13	52.61	44	3	1097.81	44	5	116.053	44	5	144.417	44

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
openser-cve-2006-6749-17-fixed	6	9.409	24	3	22.967	24	3	22.998	24	2	71.119	24	3	68.713	24	3	83.927	24
openser-cve-2006-6749-18-fixed	4	7.785	24	3	23.181	24	3	22.939	24	2	70.865	24	3	69.241	24	3	84.042	24
openser-cve-2006-6749-19		—		36	487.706	94	34	472.411	94		—			—			—	
openser-cve-2006-6749-19-fixed	12	30.069	41	6	36.255	41	6	36.468	41	2	2478.7	41		—			—	
openser-cve-2006-6749-1-fixed	4	16.229	38	4	34.831	38	4	34.828	38	2	846.495	38	3	1236.21	38	3	1252.74	38
openser-cve-2006-6749-2	18	118.552	99	16	93.117	99	16	93.23	99		—		12	178.711	99	11	224.053	99
openser-cve-2006-6749-20-fixed	4	11.762	41	4	29.634	41	4	29.501	41	2	2475.46	41		—			—	
openser-cve-2006-6749-21	65	222.085	54	12	67.067	54	12	95.167	54		—		9	860.206	54	7	705.516	54
openser-cve-2006-6749-21-fixed	12	23.892	38	6	31.871	38	6	31.636	38	2	343.022	38	3	471.473	38	3	487.04	38
openser-cve-2006-6749-22-fixed	4	9.971	38	4	27.422	38	4	27.297	38	2	344.304	38	3	470.662	38	3	487.291	38
openser-cve-2006-6749-23	200	1431.12	50	30	197.52	80	33	188.92	80		—			—			—	
openser-cve-2006-6749-23-fixed	6	11.538	29	3	23.792	29	3	23.635	29	2	135.127	29	3	152.294	29	3	167.833	29
openser-cve-2006-6749-24-fixed	4	9.006	29	3	23.739	29	3	23.796	29	2	134.752	29	3	152.327	29	3	167.04	29
openser-cve-2006-6749-25	35	75.949	48	8	28.318	48	8	28.247	48		—		4	96.969	48	4	112.429	48
openser-cve-2006-6749-25-fixed	6	9.686	26	3	23.657	26	3	23.507	26	2	40.156	26	3	38.497	26	3	53.67	26
openser-cve-2006-6749-26-fixed	4	7.74	26	3	23.831	26	3	23.717	26	2	39.923	26	3	38.499	26	3	54.048	26
openser-cve-2006-6749-27	200	2566.97	57	28	195.807	92	28	185.197	92		—		18	2432.55	75	11	2439.12	75
openser-cve-2006-6749-27-fixed	10	19.018	33	5	30.48	33	5	30.128	33	2	211.86	33	3	237.554	33	3	253.685	33
openser-cve-2006-6749-28-fixed	4	9.937	33	4	28.561	33	4	28.432	33	2	211.284	33	3	238.599	33	3	254.454	33
openser-cve-2006-6749-29	74	224.045	50	11	46.608	50	11	46.226	50		—		7	150.038	50	5	143.473	50
openser-cve-2006-6749-29-fixed	10	16.376	32	5	28.179	32	5	28.061	32	2	89.955	32	3	97.159	32	3	111.654	32
openser-cve-2006-6749-2-fixed	29	460.094	128	20	296.13	128	20	290.491	128		—			—			—	
openser-cve-2006-6749-30-fixed	4	9.374	32	4	26.337	32	4	26.331	32	2	89.992	32	3	97.295	32	3	112.287	32
openser-cve-2006-6749-31	136	1001.58	49	14	64.48	49	14	64.923	49	3	3340.83	49	7	164.97	49	5	178.526	49
openser-cve-2006-6749-31-fixed	6	10.295	27	3	23.491	27	3	23.439	27	2	83.681	27	3	85.944	27	3	101.003	27
openser-cve-2006-6749-32-fixed	4	8.337	27	3	23.386	27	3	23.44	27	2	84.207	27	3	86.191	27	3	101.696	27
openser-cve-2006-6749-33-fixed	12	34.238	39	6	37.063	39	6	37.182	39	2	890.722	39	3	1278.34	39	3	1277.27	39
openser-cve-2006-6749-34-fixed	4	12.657	39	4	31.332	39	4	31.043	39	2	895.069	39	3	1250.9	39	3	1273.64	39
openser-cve-2006-6749-35	107	2155.06	111	19	188.865	111	20	165.955	111		—			—			—	
openser-cve-2006-6749-35-fixed	12	24.83	36	6	34.351	36	6	34.237	36	2	173.902	36	3	213.938	36	3	228.151	36
openser-cve-2006-6749-36-fixed	4	9.606	35	4	26.81	35	4	27.128	35	2	189.431	35	3	226.784	35	3	242.3	35
openser-cve-2006-6749-37	200	1560.56	46	31	221.701	78	30	206.592	78		—			—			—	
openser-cve-2006-6749-37-fixed	6	11.504	27	3	23.007	27	3	22.817	27	2	104.933	27	3	108.755	27	3	124.062	27
openser-cve-2006-6749-38-fixed	4	8.654	26	3	23.529	26	3	23.707	26	2	101.945	26	3	104.531	26	3	119.653	26
openser-cve-2006-6749-39	35	68.456	46	8	30.843	46	8	30.677	46		—		4	165.025	46	4	179.867	46
openser-cve-2006-6749-39-fixed	6	9.514	24	3	23.462	24	3	23.527	24	2	36.575	24	3	33.821	24	3	48.981	24
openser-cve-2006-6749-3-fixed	32	523.268	132	20	248.317	132	20	251.812	132		—			—			—	
openser-cve-2006-6749-40-fixed	4	7.61	23	3	23.923	23	3	23.705	23	2	34.611	23	3	31.675	23	3	46.829	23

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
openser-cve-2006-6749-41	200	1399.72	51	26	377.124	101	39	750.602	101		—			—			—	
openser-cve-2006-6749-41-fixed	10	22.012	34	5	30.842	34	5	30.621	34	2	272.796	34	3	309.815	34	3	329.149	34
openser-cve-2006-6749-42-fixed	4	11.034	34	4	28.132	34	4	27.932	34	2	270.11	34	3	310.922	34	3	324.208	34
openser-cve-2006-6749-43	48	71.557	66	11	59.906	66	11	60.31	66		—		4	2223.2	66	4	2238.09	66
openser-cve-2006-6749-43-fixed	10	17.162	32	5	28.919	32	5	29.136	32	2	95.567	32	3	103.148	32	3	118.83	32
openser-cve-2006-6749-44-fixed	4	9.813	30	4	27.062	30	4	27.192	30	2	69.237	30	3	69.89	30	3	86.476	30
openser-cve-2006-6749-45	200	946.076	42	25	154.106	74	25	159.224	74		—			—			—	
openser-cve-2006-6749-45-fixed	6	10.509	27	3	23.401	27	3	23.62	27	2	83.609	27	3	85.879	27	3	100.741	27
openser-cve-2006-6749-46-fixed	4	8.211	25	3	23.379	25	3	23.408	25	2	72.288	25	3	70.744	25	3	86.29	25
openser-cve-2006-6749-4-fixed	29	498.156	131	20	357.804	131	21	326.427	131		—			—			—	
openser-cve-2006-6749-5	200	2709.89	61	46	919.906	101	57	902.404	101		—			—			—	
openser-cve-2006-6749-5-fixed	12	27.934	38	6	36.17	38	6	36.592	38	2	833.223	38	3	1139.79	38	3	1155.33	38
openser-cve-2006-6749-6-fixed	4	11.716	38	4	30.396	38	4	30.325	38	2	826.532	38	3	1133.04	38	3	1184.44	38
openser-cve-2006-6749-7	72	197.263	50	14	69.724	50	16	67.772	50		—		7	218.977	50	5	201.948	50
openser-cve-2006-6749-7-fixed	12	19.84	35	6	30.558	35	6	30.502	35	2	187.182	35	3	221.771	35	3	240.299	35
openser-cve-2006-6749-8-fixed	4	9.524	35	4	26.736	35	4	26.526	35	2	187.747	35	3	222.537	35	3	237.511	35
openser-cve-2006-6749-9	200	1132.87	45	29	160.71	75	31	181.869	75		—			—			—	
openser-cve-2006-6749-9-fixed	6	10.497	26	3	23.429	26	3	23.417	26	2	101.471	26	3	102.62	26	3	117.736	26
samba-cve-2007-0453-1-fixed	1	0.046	2	1	0.045	2	1	0.045	2	1	0.058	2	1	0.06	2	1	0.055	2
samba-cve-2007-0453-2	9	1.996	25	7	1.754	25	7	1.812	25	4	10.376	25	6	6.353	25	6	6.373	25
samba-cve-2007-0453-2-fixed	2	0.11	4	2	0.109	4	2	0.107	4	2	0.122	4	2	0.134	4	2	0.131	4
sendmail-cve-1999-0047-10-1	2	0.114	5	2	0.115	5	2	0.116	5	2	0.135	5	2	0.139	5	2	0.138	5
sendmail-cve-1999-0047-10-1-fixed	3	0.274	7	3	0.288	7	3	0.28	7	2	0.142	7	2	0.157	7	2	0.149	7
sendmail-cve-1999-0047-10-2-fixed	2	0.155	12	2	0.157	12	2	0.154	12	2	0.254	12	2	0.277	12	2	0.266	12
sendmail-cve-1999-0047-10-fixed	2	0.138	9	2	0.14	9	2	0.14	9	2	0.21	9	2	0.232	9	2	0.228	9
sendmail-cve-1999-0047-11	3	1.438	21	3	1.453	21	3	1.42	21	3	2.407	21	3	2.217	21	3	2.229	21
sendmail-cve-1999-0047-11-1-fixed	2	0.233	14	2	0.233	14	2	0.231	14	2	0.269	14	2	0.329	14	2	0.328	14
sendmail-cve-1999-0047-11-2-fixed	3	2.672	34	3	2.792	34	3	2.785	34	3	4.484	34	3	5.935	34	3	6.087	34
sendmail-cve-1999-0047-11-3-fixed	3	1.319	21	3	1.407	21	3	1.375	21	2	0.747	21	2	1.246	21	2	1.258	21
sendmail-cve-1999-0047-11-fixed	3	1.205	19	3	1.284	19	3	1.3	19	2	0.911	19	2	1.336	19	2	1.335	19
sendmail-cve-1999-0047-12	3	0.611	15	3	0.61	15	3	0.609	15	3	0.992	15	3	0.768	15	3	0.77	15
sendmail-cve-1999-0047-12-1-fixed	2	0.141	8	2	0.142	8	2	0.142	8	2	0.159	8	2	0.165	8	2	0.164	8
sendmail-cve-1999-0047-12-2-fixed	3	1.363	28	3	1.4	28	3	1.384	28	3	2.842	28	3	3.002	28	3	2.93	28
sendmail-cve-1999-0047-12-3-fixed	3	0.627	15	3	0.699	15	3	0.772	15	2	0.431	15	2	0.489	15	2	0.49	15
sendmail-cve-1999-0047-12-fixed	3	0.559	13	3	0.606	13	3	0.602	13	2	0.351	13	2	0.379	13	2	0.369	13
sendmail-cve-1999-0047-13	5	0.645	12	5	0.654	12	5	0.682	12	4	0.812	12	4	0.602	12	4	0.62	12
sendmail-cve-1999-0047-13-1-fixed	3	0.396	9	3	0.429	9	3	0.426	9	2	0.205	9	2	0.236	9	2	0.226	9
sendmail-cve-1999-0047-13-fixed	4	0.538	12	4	0.628	12	4	0.611	12	3	0.42	12	3	0.349	12	3	0.342	12

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k=7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
sendmail-cve-1999-0047-14	2	3.084	32	2	3.074	32	2	3.084	32	2	30.686	32	2	74.3	32	2	74.416	32
sendmail-cve-1999-0047-14-1-fixed	2	1.376	25	2	1.359	25	2	1.362	25	2	1.928	25	2	3.18	25	2	3.231	25
sendmail-cve-1999-0047-14-2	4	10.03	37	4	10.25	37	4	10.255	37	3	78.188	37	4	22.126	37	4	22.105	37
sendmail-cve-1999-0047-14-2-fixed	3	4.717	30	3	4.811	30	3	4.865	30	2	4.379	30	2	9.19	30	2	9.36	30
sendmail-cve-1999-0047-14-3-fixed	2	2.029	32	2	2.003	32	2	2.005	32	2	4.488	32	2	8.606	32	2	8.666	32
sendmail-cve-1999-0047-14-4-fixed	3	34.618	58	3	34.139	58	3	33.912	58	2	65.446	58	2	139.817	58	2	140.265	58
sendmail-cve-1999-0047-14-5	10	63.476	46	9	48.441	46	9	48.967	46	4	917.862	46	6	82.728	46	6	82.408	46
sendmail-cve-1999-0047-14-5-fixed	2	3.045	35	2	3.011	35	2	3.018	35	2	11.299	35	2	22.682	35	2	22.946	35
sendmail-cve-1999-0047-14-6-fixed	2	3.056	37	2	3.081	37	2	3.074	37	2	11.815	37	2	24.364	37	2	24.619	37
sendmail-cve-1999-0047-14-7	1	0.251	2	1	0.256	2	1	0.256	2	1	0.264	2	1	0.28	2	1	0.265	2
sendmail-cve-1999-0047-14-7-fixed	3	26.996	71	3	27.483	71	3	27.229	71	2	644.338	71	2	917.868	71	2	918.183	71
sendmail-cve-1999-0047-14-8	2	2.721	35	2	2.712	35	2	2.715	35	2	8.149	35	2	16.447	35	2	16.378	35
sendmail-cve-1999-0047-14-8-fixed	2	3.167	38	2	3.163	38	2	3.2	38	2	14.09	38	2	28.185	38	2	28.201	38
sendmail-cve-1999-0047-14-fixed	2	3.245	35	2	3.305	35	2	3.307	35	2	102.189	35	2	262.988	35	2	262.607	35
sendmail-cve-1999-0047-15	2	0.698	18	2	0.697	18	2	0.703	18	2	1.317	18	2	1.283	18	2	1.271	18
sendmail-cve-1999-0047-15-1-fixed	2	0.309	11	2	0.304	11	2	0.306	11	2	0.341	11	2	0.352	11	2	0.36	11
sendmail-cve-1999-0047-15-2-fixed	3	2.702	31	3	2.697	31	3	2.814	31	3	5.792	31	3	5.8	31	3	5.847	31
sendmail-cve-1999-0047-15-3	2	0.533	14	2	0.532	14	2	0.532	14	2	0.648	14	2	0.658	14	2	0.643	14
sendmail-cve-1999-0047-15-3-fixed	3	1.134	16	3	1.106	16	3	1.121	16	2	0.772	16	2	0.741	16	2	0.742	16
sendmail-cve-1999-0047-15-4-fixed	2	0.493	18	2	0.494	18	2	0.505	18	2	0.806	18	2	0.944	18	2	0.94	18
sendmail-cve-1999-0047-15-5-fixed	3	7.483	44	3	7.611	44	3	7.574	44	2	14.4	44	2	14.879	44	2	14.925	44
sendmail-cve-1999-0047-15-6	11	11.185	30	9	9.316	30	9	9.252	30	4	13.223	30	10	13.475	39	10	14.687	39
sendmail-cve-1999-0047-15-6-fixed	2	0.613	21	2	0.607	21	2	0.618	21	2	2.201	21	2	2.097	21	2	2.096	21
sendmail-cve-1999-0047-15-7-fixed	2	0.814	23	2	0.822	23	2	0.829	23	2	2.301	23	2	2.386	23	2	2.367	23
sendmail-cve-1999-0047-15-8	1	0.167	2	1	0.176	2	1	0.169	2	1	0.186	2	1	0.187	2	1	0.187	2
sendmail-cve-1999-0047-15-8-fixed	3	10.418	57	3	10.467	57	3	10.533	57	2	372.969	57	2	361.955	57	2	362.973	57
sendmail-cve-1999-0047-15-9	2	0.75	21	2	0.756	21	2	0.774	21	2	1.915	21	2	1.773	21	2	1.807	21
sendmail-cve-1999-0047-15-9-fixed	2	0.845	24	2	0.835	24	2	0.831	24	2	2.425	24	2	2.413	24	2	2.438	24
sendmail-cve-1999-0047-15-fixed	2	0.65	21	2	0.654	21	2	0.652	21	2	2.555	21	2	2.408	21	2	2.408	21
sendmail-cve-1999-0047-16	2	0.185	8	2	0.186	8	2	0.185	8	2	0.202	8	2	0.205	8	2	0.204	8
sendmail-cve-1999-0047-16-1-fixed	3	0.551	11	3	0.556	11	3	0.579	11	2	0.303	11	2	0.328	11	2	0.321	11
sendmail-cve-1999-0047-16-2-fixed	2	0.254	13	2	0.247	13	2	0.25	13	2	0.454	13	2	0.507	13	2	0.505	13
sendmail-cve-1999-0047-16-3	1	0.082	2	1	0.08	2	1	0.079	2	1	0.086	2	1	0.087	2	1	0.085	2
sendmail-cve-1999-0047-16-3-fixed	2	0.403	20	2	0.421	20	2	0.41	20	2	0.81	20	2	0.891	20	2	0.904	20
sendmail-cve-1999-0047-16-fixed	3	0.627	22	3	0.613	22	3	0.612	22	3	1.598	22	3	1.374	22	3	1.376	22
sendmail-cve-1999-0047-17	10	24.76	38	9	21.398	38	9	21.683	38	5	537.01	47	10	65.817	38	10	65.958	38
sendmail-cve-1999-0047-17-1-fixed	2	0.69	20	2	0.685	20	2	0.688	20	2	0.939	20	2	1.171	20	2	1.178	20
sendmail-cve-1999-0047-17-2	2	1.158	23	2	1.15	23	2	1.172	23	2	1.402	23	2	1.857	23	2	1.886	23

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
sendmail-cve-1999-0047-17-2-fixed	3	2.37	25	3	2.404	25	3	2.362	25	2	2.158	25	2	2.874	25	2	2.923	25
sendmail-cve-1999-0047-17-3-fixed	2	1.054	27	2	1.05	27	2	1.068	27	2	1.868	27	2	3.153	27	2	3.191	27
sendmail-cve-1999-0047-17-4-fixed	3	11.776	53	3	11.995	53	3	12.021	53	2	23.94	53	2	38.503	53	2	38.648	53
sendmail-cve-1999-0047-17-5-fixed	2	1.957	32	2	1.949	32	2	1.953	32	2	4.558	32	2	6.833	32	2	6.84	32
sendmail-cve-1999-0047-17-fixed	2	1.379	29	2	1.384	29	2	1.375	29	2	8.934	29	2	18.89	29	2	18.987	29
sendmail-cve-1999-0047-18	10	7.101	28	9	6.448	28	9	6.494	28	5	30.882	37	7	6.829	28	8	7.561	28
sendmail-cve-1999-0047-18-1-fixed	2	0.22	10	2	0.232	10	2	0.221	10	2	0.25	10	2	0.256	10	2	0.255	10
sendmail-cve-1999-0047-18-2-fixed	3	1.894	30	3	2.232	30	3	2.247	30	3	4.519	30	3	4.352	30	3	4.371	30
sendmail-cve-1999-0047-18-3	2	0.341	13	2	0.334	13	2	0.341	13	2	0.496	13	2	0.511	13	2	0.485	13
sendmail-cve-1999-0047-18-3-fixed	3	0.888	15	3	0.9	15	3	0.902	15	2	0.588	15	2	0.627	15	2	0.632	15
sendmail-cve-1999-0047-18-4-fixed	2	0.402	17	2	0.406	17	2	0.399	17	2	0.589	17	2	0.623	17	2	0.63	17
sendmail-cve-1999-0047-18-5-fixed	3	4.617	43	3	4.658	43	3	4.678	43	2	12.151	43	2	12.595	43	2	12.576	43
sendmail-cve-1999-0047-18-6-fixed	2	0.509	22	2	0.506	22	2	0.504	22	2	1.446	22	2	1.47	22	2	1.478	22
sendmail-cve-1999-0047-18-fixed	2	0.5	19	2	0.501	19	2	0.512	19	2	1.301	19	2	1.296	19	2	1.291	19
sendmail-cve-1999-0047-19-1	2	0.164	8	2	0.166	8	2	0.165	8	2	0.192	8	2	0.202	8	2	0.198	8
sendmail-cve-1999-0047-19-1-fixed	3	0.432	10	3	0.448	10	3	0.467	10	2	0.239	10	2	0.261	10	2	0.262	10
sendmail-cve-1999-0047-19-2-fixed	2	0.237	13	2	0.259	13	2	0.235	13	2	0.396	13	2	0.446	13	2	0.449	13
sendmail-cve-1999-0047-19-fixed	3	0.393	16	3	0.387	16	3	0.391	16	3	0.908	16	3	0.652	16	3	0.663	16
sendmail-cve-1999-0047-2	3	1.123	19	3	1.141	19	3	1.143	19	3	1.572	19	3	1.546	19	3	1.569	19
sendmail-cve-1999-0047-2-1-fixed	1	0.042	1	1	0.042	1	1	0.041	1	1	0.049	1	1	0.052	1	1	0.051	1
sendmail-cve-1999-0047-2-2-fixed	5	1.495	20	4	1.205	20	4	1.211	20	2	0.631	20	2	0.837	20	2	0.853	20
sendmail-cve-1999-0047-2-3	6	1.91	22	6	2.017	22	6	1.973	22	3	1.679	22	3	1.617	22	3	1.626	22
sendmail-cve-1999-0047-2-3-fixed	3	0.923	18	3	0.941	18	3	0.987	18	2	0.525	18	2	0.671	18	2	0.684	18
sendmail-cve-1999-0047-2-fixed	3	0.936	16	3	0.942	16	3	0.915	16	2	0.347	16	2	0.494	16	2	0.499	16
sendmail-cve-1999-0047-3	5	0.755	16	5	0.743	16	5	0.754	16	3	0.792	16	4	0.811	16	4	0.788	16
sendmail-cve-1999-0047-3-1-fixed	1	0.036	1	1	0.036	1	1	0.035	1	1	0.044	1	1	0.05	1	1	0.046	1
sendmail-cve-1999-0047-3-2-fixed	5	0.667	14	4	0.675	14	4	0.588	14	2	0.284	14	2	0.292	14	2	0.289	14
sendmail-cve-1999-0047-3-3	5	0.737	16	5	0.756	16	5	0.751	16	3	0.789	16	4	0.811	16	4	0.783	16
sendmail-cve-1999-0047-3-3-fixed	3	0.444	12	3	0.463	12	3	0.464	12	2	0.207	12	2	0.232	12	2	0.236	12
sendmail-cve-1999-0047-3-fixed	3	0.372	10	3	0.396	10	3	0.393	10	2	0.181	10	2	0.199	10	2	0.194	10
sendmail-cve-1999-0047-4	3	0.262	8	3	0.263	8	3	0.353	8	3	0.322	8	3	0.318	8	3	0.565	8
sendmail-cve-1999-0047-4-1	5	0.461	11	5	0.464	11	5	0.492	11	3	0.416	11	4	0.458	11	4	0.495	11
sendmail-cve-1999-0047-4-1-fixed	3	0.302	8	3	0.319	8	3	0.31	8	2	0.136	8	2	0.166	8	2	0.153	8
sendmail-cve-1999-0047-4-fixed	3	0.254	6	3	0.26	6	3	0.26	6	2	0.116	6	2	0.124	6	2	0.125	6
sendmail-cve-1999-0047-5	2	2.248	29	2	2.24	29	2	2.268	29	2	10.582	29	2	24.484	29	2	24.538	29
sendmail-cve-1999-0047-5-1-fixed	1	0.085	1	1	0.084	1	1	0.085	1	1	0.1	1	1	0.102	1	1	0.105	1
sendmail-cve-1999-0047-5-2-fixed	5	7.174	31	4	5.559	31	4	5.582	31	2	4.637	31	2	10.489	31	2	10.504	31
sendmail-cve-1999-0047-5-3	4	6.953	34	4	7.044	34	4	7.03	34	3	26.84	34	4	13.85	34	4	13.944	34

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
sendmail-cve-1999-0047-5-3-fixed	3	3.188	27	3	3.23	27	3	3.215	27	2	2.46	27	2	3.732	27	2	3.763	27
sendmail-cve-1999-0047-5-4-fixed	1	0.085	1	1	0.086	1	1	0.085	1	1	0.098	1	1	0.104	1	1	0.109	1
sendmail-cve-1999-0047-5-5-fixed	6	7.844	35	5	6.733	35	5	6.79	35	2	7.582	35	2	14.407	35	2	14.474	35
sendmail-cve-1999-0047-5-6	5	12.157	38	5	11.673	38	5	11.855	38	3	38.849	38	3	16.218	38	3	16.247	38
sendmail-cve-1999-0047-5-6-fixed	3	4.268	31	3	4.346	31	3	4.336	31	2	5.148	31	2	10.687	31	2	10.824	31
sendmail-cve-1999-0047-5-7-fixed	1	0.085	1	1	0.085	1	1	0.085	1	1	0.105	1	1	0.107	1	1	0.109	1
sendmail-cve-1999-0047-5-8-fixed	8	15.102	39	5	8.651	39	5	8.78	39	2	16.825	39	2	32.138	39	2	32.484	39
sendmail-cve-1999-0047-5-9	2	2.068	32	2	2.061	32	2	2.062	32	2	3.401	32	2	5.565	32	2	5.635	32
sendmail-cve-1999-0047-5-9-fixed	4	8.165	34	4	7.736	34	4	7.803	34	2	7.367	34	2	14.816	34	2	14.802	34
sendmail-cve-1999-0047-5-fixed	4	7.622	31	3	5.487	31	3	5.457	31	2	21.419	31	2	53.429	31	2	53.605	31
sendmail-cve-1999-0047-6	2	0.496	15	2	0.49	15	2	0.487	15	2	0.724	15	2	0.683	15	2	0.7	15
sendmail-cve-1999-0047-6-1-fixed	1	0.061	1	1	0.061	1	1	0.06	1	1	0.082	1	1	0.093	1	1	0.081	1
sendmail-cve-1999-0047-6-2-fixed	5	1.563	17	4	1.335	17	4	1.254	17	2	0.67	17	2	0.748	17	2	0.759	17
sendmail-cve-1999-0047-6-3	2	0.33	12	2	0.329	12	2	0.435	12	2	0.461	12	2	0.446	12	2	0.444	12
sendmail-cve-1999-0047-6-3-fixed	3	0.707	13	3	0.737	13	3	0.734	13	2	0.421	13	2	0.434	13	2	0.433	13
sendmail-cve-1999-0047-6-4-fixed	1	0.06	1	1	0.061	1	1	0.061	1	1	0.077	1	1	0.086	1	1	0.083	1
sendmail-cve-1999-0047-6-5-fixed	6	1.992	21	5	1.79	21	5	1.769	21	2	1.09	21	2	1.159	21	2	1.184	21
sendmail-cve-1999-0047-6-6	3	1.357	22	3	1.47	22	3	1.355	22	3	2.238	22	3	1.78	22	3	1.849	22
sendmail-cve-1999-0047-6-6-fixed	3	1.039	17	3	1.091	17	3	1.102	17	2	0.755	17	2	0.798	17	2	0.806	17
sendmail-cve-1999-0047-6-7-fixed	1	0.06	1	1	0.061	1	1	0.061	1	1	0.079	1	1	0.08	1	1	0.081	1
sendmail-cve-1999-0047-6-8-fixed	8	4.314	25	5	2.662	25	5	2.647	25	2	2.412	25	2	2.409	25	2	2.413	25
sendmail-cve-1999-0047-6-9	2	0.527	18	2	0.648	18	2	0.526	18	2	0.771	18	2	0.895	18	2	1.022	18
sendmail-cve-1999-0047-6-9-fixed	4	2.128	20	4	2.228	20	4	2.193	20	2	1.05	20	2	1.073	20	2	1.08	20
sendmail-cve-1999-0047-6-fixed	4	1.68	17	3	1.038	17	3	1.023	17	2	0.87	17	2	0.827	17	2	0.846	17
sendmail-cve-1999-0047-7	1	0.057	1	1	0.058	1	1	0.058	1	1	0.063	1	1	0.066	1	1	0.063	1
sendmail-cve-1999-0047-7-1-fixed	3	0.353	8	3	0.369	8	3	0.365	8	2	0.19	8	2	0.191	8	2	0.197	8
sendmail-cve-1999-0047-7-2-fixed	3	0.346	8	3	0.358	8	3	0.357	8	2	0.181	8	2	0.195	8	2	0.185	8
sendmail-cve-1999-0047-7-3	1	0.057	1	1	0.057	1	1	0.056	1	1	0.063	1	1	0.061	1	1	0.065	1
sendmail-cve-1999-0047-7-3-fixed	2	0.219	16	2	0.223	16	2	0.228	16	2	0.429	16	2	0.457	16	2	0.473	16
sendmail-cve-1999-0047-7-fixed	2	0.251	12	2	0.202	12	2	0.208	12	2	0.448	12	2	0.379	12	2	0.438	12
sendmail-cve-1999-0047-8	3	3.761	30	3	3.769	30	3	3.744	30	3	9.905	30	3	7.87	30	3	7.909	30
sendmail-cve-1999-0047-8-1-fixed	1	0.062	1	1	0.062	1	1	0.062	1	1	0.073	1	1	0.078	1	1	0.079	1
sendmail-cve-1999-0047-8-2-fixed	5	4.11	26	4	3.237	26	4	3.252	26	2	1.852	26	2	3.189	26	2	3.25	26
sendmail-cve-1999-0047-8-3	2	0.777	21	2	0.778	21	2	0.768	21	2	0.995	21	2	1.059	21	2	1.093	21
sendmail-cve-1999-0047-8-3-fixed	3	1.616	22	3	1.987	22	3	1.858	22	2	0.964	22	2	1.319	22	2	1.346	22
sendmail-cve-1999-0047-8-4-fixed	1	0.062	1	1	0.061	1	1	0.062	1	1	0.081	1	1	0.079	1	1	0.081	1
sendmail-cve-1999-0047-8-5-fixed	6	5.079	30	5	3.952	30	5	3.921	30	2	2.408	30	2	4.024	30	2	4.081	30
sendmail-cve-1999-0047-8-6	4	3.826	26	3	2.786	26	3	2.771	26	2	1.979	26	2	2.187	26	2	2.174	26

Continued on the next page…

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for `WP`, `NewST`, `NewST` with a threshold $k = 7$, `SATQE`, `NewSP` and `NewST` +`NewSP`. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k = 7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
sendmail-cve-1999-0047-8-6-fixed	3	2.525	28	3	2.697	28	3	2.655	28	2	2.458	28	2	3.925	28	2	3.933	28
sendmail-cve-1999-0047-8-fixed	3	2.884	25	3	3.039	25	3	3.042	25	2	2.824	25	2	5.182	25	2	5.318	25
sendmail-cve-1999-0047-9	3	1.035	20	3	1.033	20	3	1.023	20	3	1.735	20	3	1.353	20	3	1.36	20
sendmail-cve-1999-0047-9-1-fixed	1	0.048	1	1	0.048	1	1	0.047	1	1	0.059	1	1	0.066	1	1	0.097	1
sendmail-cve-1999-0047-9-2-fixed	5	0.953	16	4	0.832	16	4	0.806	16	2	0.517	16	2	0.56	16	2	0.556	16
sendmail-cve-1999-0047-9-3	2	0.237	11	2	0.236	11	2	0.236	11	2	0.286	11	2	0.297	11	2	0.296	11
sendmail-cve-1999-0047-9-3-fixed	3	0.466	12	3	0.48	12	3	0.538	12	2	0.263	12	2	0.275	12	2	0.268	12
sendmail-cve-1999-0047-9-4-fixed	1	0.048	1	1	0.048	1	1	0.048	1	1	0.057	1	1	0.066	1	1	0.07	1
sendmail-cve-1999-0047-9-5-fixed	6	1.305	20	5	1.162	20	5	1.152	20	2	0.743	20	2	0.792	20	2	0.786	20
sendmail-cve-1999-0047-9-6	4	1.223	16	3	0.932	16	3	0.97	16	2	0.571	16	2	0.601	16	2	0.593	16
sendmail-cve-1999-0047-9-6-fixed	3	0.996	18	3	1.088	18	3	1.064	18	2	0.546	18	2	0.702	18	2	0.718	18
sendmail-cve-1999-0047-9-fixed	3	0.883	15	3	0.945	15	3	0.945	15	2	0.527	15	2	0.488	15	2	0.492	15
sendmail-cve-1999-0206-1	3	1.034	20	3	1.131	20	3	1.068	20	3	1.815	20	3	1.445	20	3	1.409	20
sendmail-cve-1999-0206-1-1	3	0.735	16	3	0.753	16	3	0.738	16	3	1.605	16	3	1.187	16	3	1.23	16
sendmail-cve-1999-0206-1-1-fixed	3	0.621	16	3	0.63	16	3	0.623	16	3	1.438	16	3	1.069	16	3	1.108	16
sendmail-cve-1999-0206-1-2	2	0.189	10	2	0.188	10	2	0.187	10	2	0.225	10	2	0.239	10	2	0.234	10
sendmail-cve-1999-0206-1-fixed	3	0.926	20	3	0.947	20	3	0.939	20	3	1.783	20	3	1.314	20	3	1.314	20
sendmail-cve-1999-0206-2	5	1.831	20	5	1.881	20	5	1.847	20	4	5.073	20	4	2.064	20	4	1.915	20
sendmail-cve-1999-0206-2-1	5	1.825	20	5	1.861	20	5	1.846	20	4	5.073	20	4	1.949	20	4	1.932	20
sendmail-cve-1999-0206-2-1-fixed	5	1.733	20	5	1.722	20	5	1.74	20	4	4.96	20	4	1.847	20	4	1.827	20
sendmail-cve-1999-0206-2-2	3	0.437	16	3	0.419	16	3	0.428	16	3	0.599	16	3	0.657	16	3	0.628	16
sendmail-cve-1999-0206-2-fixed	5	2.23	26	5	2.226	26	5	2.26	26	4	4.701	26	5	3.022	26	5	3.055	26
sendmail-cve-2001-0653-1	6	36.165	59	6	35.896	59	6	35.958	59	4	1747.65	59	4	502.115	59	4	503.236	59
sendmail-cve-2001-0653-3-fixed	16	21.513	51	14	19.463	51	14	19.355	51		—			—			—	
sendmail-cve-2002-0906-1	1	0.08	2	1	0.076	2	1	0.077	2	1	0.088	2	1	0.084	2	1	0.083	2
sendmail-cve-2002-0906-1-1-fixed	1	0.006	0	1	0.006	0	1	0.005	0	1	0.006	0	1	0.006	0	1	0.006	0
sendmail-cve-2002-0906-2	1	0.076	2	1	0.077	2	1	0.076	2	1	0.087	2	1	0.089	2	1	0.085	2
sendmail-cve-2002-0906-2-1-fixed	3	0.315	12	3	0.316	12	3	0.316	12	2	0.214	12	2	0.247	12	2	0.246	12
sendmail-cve-2002-1337-1	3	0.359	13	3	0.36	13	3	0.356	13	3	0.517	13	3	0.52	13	3	0.503	13
sendmail-cve-2002-1337-1-fixed	7	1.579	22	5	1.301	22	5	1.276	22	4	2.512	22	6	1.776	22	6	1.783	22
sendmail-cve-2002-1337-2	3	0.515	16	3	0.518	16	3	0.521	16	3	0.776	16	3	0.808	16	3	0.791	16
sendmail-cve-2002-1337-2-fixed	6	2.208	28	5	1.979	28	5	1.99	28	4	6.19	28	6	3.23	28	6	3.318	28
sendmail-cve-2002-1337-3	3	0.46	13	3	0.459	13	3	0.45	13	3	0.627	13	3	0.618	13	3	0.636	13
sendmail-cve-2002-1337-3-fixed	7	4.257	29	6	4.569	29	6	4.565	29	4	9.22	29	6	5.922	29	6	5.944	29
sendmail-cve-2003-0161-1	4	18.358	29	4	17.451	29	4	17.737	29	3	10.36	29	3	10.68	29	3	10.675	29
sendmail-cve-2003-0161-1-1-fixed	1	0.044	1	1	0.044	1	1	0.043	1	1	0.054	1	1	0.054	1	1	0.053	1
sendmail-cve-2003-0161-1-2	4	19.078	32	4	18.625	32	4	18.871	32	3	34.309	32	3	18.838	32	3	19.213	32
sendmail-cve-2003-0161-1-2-fixed	4	3.555	32	4	3.598	32	4	3.576	32	3	16.336	32	3	12.452	32	3	12.481	32

Continued on the next page. . .

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

Table A.1 – Continued

	WP			NewST			NewST, $k=7$			SATQE			NewSP			NewST + NewSP		
Benchmark	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**	**Iterations**	**Time**	**Predicates**
sendmail-cve-2003-0161-1-fixed	4	3.383	29	4	3.484	29	4	3.475	29	3	6.578	29	3	7.136	29	3	7.207	29
sendmail-cve-2003-0161-2	4	16.773	25	4	16.297	25	4	16.226	25	3	6.101	25	3	6.1	25	3	6.17	25
sendmail-cve-2003-0161-2-1-fixed	1	0.043	1	1	0.043	1	1	0.042	1	1	0.053	1	1	0.052	1	1	0.053	1
sendmail-cve-2003-0161-2-2	4	17.999	28	4	17.763	28	4	18.121	28	3	13.472	28	3	8.946	28	3	9.161	28
sendmail-cve-2003-0161-2-2-fixed	4	3.382	28	4	3.442	28	4	3.495	28	3	8.909	28	3	6.771	28	3	6.839	28
sendmail-cve-2003-0161-2-fixed	4	3.198	25	4	3.222	25	4	3.24	25	3	4.622	25	3	4.712	25	3	4.716	25
sendmail-cve-2003-0681-1	5	2.107	33	5	2.189	33	5	2.152	33	4	393.043	33	4	16.321	33	4	16.416	33
sendmail-cve-2003-0681-2	22	57.63	94	14	34.267	46	14	34.57	46		—		9	47.108	46	6	38.466	46
sendmail-cve-2003-0681-2-1-fixed	24	41.035	75	15	32.062	75	15	31.446	75		—						—	
sendmail-cve-2003-0681-2-fixed	9	19.069	58	8	14.626	58	8	14.599	58		—		6	143.234	58	6	143.646	58
spamassassin-bid-6679	9	4.722	42	8	4.52	42	8	4.457	42	4	2286.49	42	7	181.681	42	7	184.709	42
spamassassin-bid-6679-fixed	9	4.079	42	10	4.792	42	10	4.805	42	4	2280.38	42	7	181.77	42	7	182.808	42
wu-ftpd-cve-1999-0368-1	11	12.604	54	9	15.843	54	9	15.482	54		—		8	60.707	54	8	66.097	54
wu-ftpd-cve-1999-0368-1-fixed	2	0.139	4	2	0.138	4	2	0.138	4	2	0.152	4	2	0.152	4	2	0.157	4
wu-ftpd-cve-1999-0368-2-1-fixed	49	728.709	87	23	380.112	87	26	355.048	87		—		59	1951.3	87	24	1415.92	87
wu-ftpd-cve-1999-0368-2-2-fixed	49	728.804	87	23	383.291	87	26	354.852	87		—		59	1945.43	87	24	1371.2	87
wu-ftpd-cve-1999-0368-2-fixed	49	728.991	87	23	383.924	87	26	357.222	87		—		59	1946.88	87	24	1464.59	87
wu-ftpd-cve-1999-0368-3-1-fixed	56	554.155	87	22	283.851	87	27	305.812	87		—		66	1639.01	87	26	1286.73	87
wu-ftpd-cve-1999-0368-3-2-fixed	56	554.398	87	22	285.504	87	27	305.157	87		—		66	1632.43	87	26	1188.19	87
wu-ftpd-cve-1999-0368-3-fixed	56	555.476	87	22	286.222	87	27	306.265	87		—		66	1631.39	87	26	1188.01	87
wu-ftpd-cve-1999-0368-4	11	15.292	57	9	16.291	57	9	16.057	57		—		11	25.654	57	10	23.691	57
wu-ftpd-cve-1999-0368-4-1	11	15.535	57	9	16.196	57	9	16.066	57		—		11	25.513	57	10	23.697	57
wu-ftpd-cve-1999-0368-4-1-fixed	53	286.014	87	24	214.71	87	28	192.824	87		—		65	1281.39	87	25	1046.47	87
wu-ftpd-cve-1999-0368-4-2	11	15.615	57	9	16.088	57	9	16.266	57		—		11	25.663	57	10	23.5	57
wu-ftpd-cve-1999-0368-4-2-fixed	53	288.51	87	24	214.043	87	28	192.747	87		—		65	1289.95	87	25	1044.41	87
wu-ftpd-cve-1999-0368-4-fixed	53	285.763	87	24	214.838	87	28	192.908	87		—		65	1284.85	87	25	1045.64	87
wu-ftpd-cve-1999-0368-5-1-fixed	46	625.718	87	24	380.282	87	30	426.899	87		—		44	1403.96	87	26	1105.33	87
wu-ftpd-cve-1999-0368-5-fixed	46	630.845	87	24	381.699	87	30	420.927	87		—		44	1372.28	87	26	1109.7	87
wu-ftpd-cve-1999-0368-6	47	3049.8	158	28	2795.19	158	35	2855.89	158		—			—			—	
wu-ftpd-cve-1999-0368-6-1	47	3078.09	158	28	2808.09	158	35	2864.1	158		—			—			—	
wu-ftpd-cve-1999-0368-6-1-fixed	41	393.282	87	24	308.872	87	29	310.508	87		—		52	1168.61	87	24	1001.49	87
wu-ftpd-cve-1999-0368-6-fixed	41	392.488	87	24	308.463	87	29	310.413	87		—		52	1172.84	87	24	987.949	87
wu-ftpd-cve-1999-0368-7	19	111.369	88	17	122.083	88	17	109.871	88		—			—			—	
wu-ftpd-cve-1999-0368-7-fixed	61	1178.29	136	42	1130.97	136	45	1024.9	136		—		64	1945.86	136	39	1754.54	136

Table A.1. Ku et. al. benchmark suite [KHCL07]: detailed results for WP, NewST, NewST with a threshold $k = 7$, SATQE, NewSP and NewST +NewSP. Red color is used to show that the time limit or refinement iterations limits was reached.

```
#include <ansi-c/expr2c.h>
#include <ansi-c/c_types.h>
#include <goto-programs/string_abstraction.h>
#include <std_expr.h>
#include <expr_util.h>
#include <pointer_expr.h>
#include <string_utils.h>
#include <invariant_test.h>

class null_pointer_invariant_testt : public invariant_testt
{
public:
  null_pointer_invariant_testt(contextt &context) :
        invariant_testt("NP", "NULL-Pointer", context) {}
  virtual ~null_pointer_invariant_testt(void) {}
  virtual void get_invariants(
  const loop_summaryt &summary, std::set<exprt> &potential_invariants);
};


// Purpose: Tests for pointer offset validity preservation
void null_pointer_invariant_testt::get_invariants(const loop_summaryt &summary,
         std::set<exprt> &potential_invariants)
{
  namespacet ns(context);
  stream_message_handlert mh(std::cout);
  string_abstractiont abs(context, mh);
  std::list<exprt> pointers;

  // find some pointers in loop variables
  for(std::set<exprt>::const_iterator it=summary.variant.begin();
          it!=summary.variant.end(); it++)
  {
    if(it->type().id()=="pointer" && is_char_type(it->type().subtype()))
      pointers.push_back( it);
  }

  // test the invariant for every pointer
  for(std::list<exprt>::iterator it = pointers.begin(); it!=pointers.end(); it++)
  {
    same_object_exprt invariant( it, gen_zero(it->type()));
    invariant.make_not();
    potential_invariants.insert(invariant);
  }
}
```

Figure A.1. An example of the class that extends generic `invariant_test` class to implement a domain of (non-)NULL pointer preservation invariants. Every discovered potential invariant is added to a common pool and will be checked later in program execution.

Tables

Figures

List of Algorithms

Bibliography

[ABD$^+$07] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. In Manuvir Das and Dan Grossman, editors, *Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48. ACM, 2007.

[Ale96] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996.

[AM79] Edward Ashcroft and Zohar Manna. The translation of 'go to' programs to 'while' programs. In *Classics in software engineering*, pages 49–61. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BAL09] Amir M. Ben-Amram and Chin Soon Lee. Ranking functions for size-change termination II. In *Logical Methods in Computer Science*, volume 5, chapter 8. 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BCC$^+$07] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *Principles of Programming Languages (POPL)*, POPL '07, pages 211–224, New York, NY, USA, 2007. ACM.

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

[BCDR04] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In Jensen and Podelski [JP04], pages 388–403.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

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

[BCP06] Ittai Balaban, Ariel Cohen, and Amir Pnueli. Ranking abstraction of recursive programs. In E. Emerson and Kedar Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 267–281. Springer Berlin / Heidelberg, 2006.

[BDNL02] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Software update via mobile agent based programming. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 32–36, New York, NY, USA, 2002. ACM.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2006.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[BMS05] Aaron Bradley, Zohar Manna, and Henny Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram Rajamani, editors, *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 247–250. Springer Berlin / Heidelberg, 2005.

[BPR03] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 5:49–58, 2003.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OPENSMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015, pages 150–153, Paphos, Cyprus, 2010. Springer.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report 2002-09, Microsoft Research, September 2002.

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

[BSBA07] Chiara Braghin, Natasha Sharygina, and Katerina Barone-Adesi. Automated verification of security policies in mobile code. In *Integrated Formal Methods IFM07*, volume 4591 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming (PLILP)*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[CC99] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6:69–95, 1999.

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *11th International Symposium on Static Analysis, SAS2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[CCF⁺05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *14th European Symposium on Programming (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCF⁺07] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing predicate abstractions by integrating BDDs and SMT solvers. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 69–76. IEEE Computer Society, 2007.

[CCG⁺04] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[CCG⁺08] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 148–162. Springer Berlin / Heidelberg, 2008.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[CDH⁺00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, and Zheng Hongjun Robby. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering (ICES)*, pages 439–448. ACM, 2000.

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

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Emerson and Aravinda Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[CGK$^+$02] Edmund Clarke, Anubhav Gupta, James Kukula, Ofer Strichman, Ed Brinksma, and Kim Larsen. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 695–709. Springer Berlin / Heidelberg, 2002.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGLA$^+$08] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2008.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[CH78a] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[CH78b] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 265–278, New York, NY, USA, 2011. ACM.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Jensen and Podelski [JP04], pages 168–176.

[CKRW10] Byron Cook, Daniel Kroening, Philipp Ruemmer, and Christoph Wintersteiger. Ranking function synthesis for bit-vector relations. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In P. Godefroid, editor, *Proceedings of SPIN 2005*, number 3639 in Lecture Notes in Computer Science, pages 75–90. Springer Verlag, 2005.

[CKS06] Byron Cook, Daniel Kroening, and Natasha Sharygina. Accurate theorem proving for program verification. *Leveraging Applications of Formal Methods*, pages 96–114, 2006.

[CKSY04] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[CKSY05a] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 570–574, 2005.

[CKSY05b] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 570–574. Springer, 2005.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In *Informatics — 10 Years Back, 10 Years Ahead, volume 2000 of Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[CP93] Ritu Chadha and David A. Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5/6):705–744, 1993.

[CPR05] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *International Symposium on Static Analysis (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Programming Language Design and Implementation (PLDI)*, pages 415–426. ACM, 2006.

[CPR09] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.

[CS01] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001.

[CTVW04] Edmund Clarke, Muralidhar Talupur, Helmut Veith, and Dong Wang. Sat based predicate abstraction for hardware verification. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 267–268. Springer Berlin / Heidelberg, 2004.

[CU98] Michael Colón and Tomás Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan Hu and Moshe Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer Berlin / Heidelberg, 1998.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 51, Washington, DC, USA, 2001. IEEE Computer Society.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 160–171, London, UK, 1999. Springer-Verlag.

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Proceedings of the Workshop on Advances in Verification (WAVE)*, pages 1–8, 2000.

[Don02] Mark E. Donaldson. Inside the buffer overflow attack: Mechanism, method, & prevention. The SANS' Information Security Reading Room. http://www.sans.org/, April 2002.

[DPG97] Du Dingzhu, Panos M. Pardalos, and Jun Gu, editors. *Satisfiability Problem: Theory and Applications*. American Mathematical Society, Boston, MA, USA, 1997.

[DRS03] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Programming Language Design and Implementation (PLDI)*, pages 155–167, 2003.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, December 2007.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, 2003.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer Berlin / Heidelberg, 2001.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[Fra] Frama-C. http://frama-c.com/.

[GCNR10] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Refining abstract interpretations. *Information Processing Letters*, 110(16):666–671, 2010.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40 of *SIGPLAN Notes*, pages 213–223, New York, NY, USA, 2005. ACM.

[GLAS09] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 239–251, New York, NY, USA, 2009. ACM.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security (NDSS)*. The Internet Society, 2008.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[GR06] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2006.

[GR07] Denis Gopan and Thomas W. Reps. Low-level library analysis and summarization. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007.

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

[GS05] Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 112–124, 2005.

[HH95] Thomas A. Henzinger and Pei-Hsin Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems II*, pages 265–293, London, UK, 1995. Springer-Verlag.

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

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

[HJP11] Matthias Heizmann, Neil Jones, and Andreas Podelski. Size-change termination and transition invariants. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 22–50. Springer Berlin / Heidelberg, 2011.

[Hoa69] Tony Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM (JACM)*, 50(1):63–69, 2003.

[Hol03] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[IBM07] IBM Internet Security Systems. X-Force 2006 trend statistics. `http://www.iss.net/`, January 2007.

[inn] https://www.isc.org/software/inn.

[JBS$^+$07] Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kroening, and Christoph Wintersteiger. A first step towards a unified proof checker for QBF. In *Proceedings of SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2007.

[JIGG05] Himanshu Jain, Franjo Ivancic, Aarti Gupta, and Malay K. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 397–412, 2005.

[JKSC05] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In *Design Automation Conference (DAC)*, pages 445–450. ACM, 2005.

[JM05] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2005.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 459–473, 2006.

[JP04] Kurt Jensen and Andreas Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 14–25, 2000.

[Kau98] Matt Kaufmann. ACL2 support for verification projects. In *Automated Deduction — CADE-15*, volume 1421 of *Lecture Notes in Computer Science*, pages 220–238. Springer Berlin / Heidelberg, 1998.

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

[KM73] Shmuel Katz and Zohar Manna. A heuristic approach to program verification. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 500–512, 1973.

[KS03] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[KS06] Daniel Kroening and Natasha Sharygina. Approximating predicate images for bit-vector logic. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2006.

[KSTW10] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Termination analysis with compositional transition invariants. In *International Conference on Computer-Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, Edinburgh, UK, 2010. Springer.

[Kur95] Robert Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, 1995.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis (SAS)*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[LBC05] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2005.

[Lei10] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages (POPL)*, volume 36, pages 81–92. ACM, 2001.

[LNO06] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM02] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[MFH$^+$06] Roman Manevich, John Field, Thomas A. Henzinger, Ganesan Ramalingam, and Mooly Sagiv. Abstract counterexample-based refinement for powerset domains. In *Program Analysis and Compilation (PAC)*, volume 4444 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2006.

[Min01] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.

[Min06] Antoine Miné. The octagon abstract domain. *Higher Order Symbolic Computation*, 19(1):31–100, 2006.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient Theory and Applications of Satisfiability Testing solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535. ACM, June 2001.

[Moy09] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.

[MSA] MATHSAT. `http://mathsat.itc.it`.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[PP90] Doron Peled and Amir Pnueli. Proving partial order liveness properties. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 553–571. Springer, 1990.

[PR04a] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 465–486. Springer, 2004.

[PR04b] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 32–41. IEEE Computer Society, 2004.

[PR07] Andreas Podelski and Andrey Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages (PADL)*, pages 245–259. Springer, 2007.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[RCK07] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[RSY04] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In *International Conference on Automated Deduction (CADE)*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[SAN] The SANS Institute. `http://www.sans.org/`.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 132–143, New York, NY, USA, 1977. ACM.

[SKH03] Axel Simon, Andy King, and Jacob M. Howe. Two variables per linear inequality as an abstract domain. In *Proceedings of the 12th International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR 2002)*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2003.

[SLC$^+$99] Jeff Scott, Lea Hwang Lee, Ann Chin, John Arends, and Bill Moyer. Designing the m·core^{tm} m3 cpu architecture. In *International Conference on Computer Design (ICCD)*, pages 94–101, 1999.

[snu] SNU real-time benchmarks. `http://archi.snu.ac.kr/realtime/benchmark/`.

[SP81] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. Program Flow Analysis: theory and applications. Prentice-Hall, 1981.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, 1996.

[SSM04] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *Principles of Programming Languages (POPL)*, pages 318–329. ACM, 2004.

[SSM08] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constructing invariants for hybrid systems. *Formal Methods in System Design*, 32(1):25–55, 2008.

[Tar81] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of ACM*, 28(3):594–614, 1981.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings London Mathematical Society*, 2(42):230–265, 1936.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Cambridge, 1949.

[Weg73] Ben Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 524–536, 1973.

[Yic] YICES. `http://yices.csl.sri.com`.

[Z3] Z3. `http://research.microsoft.com/projects/z3`.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *International Symposium on Foundations of Software Engineering*, pages 97–106. ACM, 2004.