FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

First-order Theorem Proving for Program Analysis and Theory Reasoning

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Ioan-Dumitru Dragan
Matrikelnummer 0856561

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Laura Kovács

Diese Dissertation haben begutachtet:

(Laura Kovács)	(Andrei Voronkov)	(Armin Biere)

Wien, 19.01.2015

(Ioan-Dumitru Dragan)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

First-order Theorem Proving for Program Analysis and Theory Reasoning

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Ioan-Dumitru Dragan
Registration Number 0856561

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Laura Kovács

The dissertation has been reviewed by:

(Laura Kovács)	(Andrei Voronkov)	(Armin Biere)

Wien, 19.01.2015

(Ioan-Dumitru Dragan)

Erklärung zur Verfassung der Arbeit

Ioan-Dumitru Dragan
Neustiftgasse 89-91/1/19, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Analyzing and verifying computer programs is an important and challenging task. Banks, hospitals, companies, organizations and individuals heavily depend on very complex computer systems, such as Internet, networking, online payment systems, and autonomous devices. These systems are integrated in an even more complicated environment, using various computer devices. Technically, software systems rely on software implementing complicated arithmetic and logical operations over the computer memory. If this software is not reliable, the costs to the economy and society can be huge. Software development practices therefore need rigorous methods ensuring that the program behaves as expected.

Formal verification provides a methodology for making reliable and robust systems, by using program properties to hold at intermediate points of the program and using these properties to prove that programs have no errors. Providing such properties manually requires a considerable amount of work by highly skilled personnel and makes verification commercially not viable. Formal verification therefore requires non-trivial automation for generating valid program properties, such as loop invariants.

In this thesis we study the use of first-order theorem proving for generating and proving program properties. Our thesis provides a fully automated tool support, called Lingva, for generating quantified invariants of programs over arrays, and shows experimentally that the generated invariants summarize the behavior of the considered loops. Our work is based on the recently introduced symbol elimination method for invariant generation, using a saturation-based first-order theorem prover.

As program properties involve both logical and arithmetical operations over unbounded data structures, such as arrays, generating and proving program properties requires efficient methods for reasoning with both theories and quantifiers. Another contribution of this thesis comes with the integration of the bound propagation method for solving systems of linear inequalities in the first-order theorem prover Vampire. Our work provides an automated tool support for using Vampire for deciding satisfiability of a system of linear inequalities over the reals or rationals. We experimentally show that bound propagation in Vampire performs well when compared to state-of-the-art satisfiability modulo theory solvers on hard linear optimization problems.

Our arithmetic solver is limited to conjunction of linear inequalities, while arithmetic program properties usually have a more complex boolean structure, using a combination of logical conjunction, disjunction and negation. To make our work applicable for handling such complex arithmetic properties, another contribution of this thesis is the integration of boolean satisfiability (SAT) solvers within Vampire. Our work exploits the recently introduced AVATAR framework for separating the first-order reasoning part of a problem from its boolean structure. We describe our technical and implementation challenges for integrating the best performing SAT solvers within Vampire, and use our implementation to evaluate the AVATAR framework on a large set of problems coming from the TPTP library of automated theorem provers.

Kurzfassung

Die Analyse und Verifikation von Computerprogrammen ist eine sowohl wichtige als auch schwierige Aufgabe. Banken, Spitäler, Firmen, Organisationen und einzelne Personen sind auf sehr komplexe Computersysteme wie das Internet, Netzwerktechnologien, elektronische Bezahlsysteme oder autonome Systeme angewiesen. Diese Systeme sind in komplexe Umgebungen mit anderen elektronischen Geräten vernetzt. Technisch gesehen basieren diese Systeme auf Software, welche komplizierte Algorithmen und Logikoperationen auf dem Speicher ausführt. Arbeitet diese Software nicht zuverlässig, können der Wirtschaft und der Gesellschaft hohe Kosten entstehen. Bei der Softwareentwicklung sind daher rigorose Methoden nötig, um sicherzustellen, dass sich die Programme wie erwartet verhalten.

Formale Verifikation bietet eine Methodik, um zuverlässige und robuste Systeme zu bauen. Ausgehend von Programmeigenschaften, die an bestimmten Programmpunkten gelten, wird die Fehlerfreiheit des Programms gezeigt. Das manuelle Definieren dieser Programmeigenschaften muss jedoch von hochqualifiziertem Personal unter hohem Zeitaufwand durchgeführt werden, wodurch Verifikation unwirtschaftlich wird. Formale Verifikation erfordert daher eine nichttriviale Automation der Generierung gültiger Programmeigenschaften wie beispielsweise Schleifeninvarianten.

In dieser Dissertation untersuchen wir, wie mit Beweismethoden der Prädikatenlogik erster Stufe Programmeigenschaften gefunden und bewiesen werden können. Die erarbeiteten Methoden wurden in einem vollständig automatisierten Tool namens Lingva umgesetzt, welches quantifizierte Invarianten über Arrays generiert. Wir zeigen experimentell, dass die generierten Invarianten das Verhalten der analysierten Schleifen zusammenfassen. Unsere Arbeit basiert auf der erst jüngst entwickelten Methode der Elimination von Symbolen bei Generierung von Invarianten, wobei ein sättigungsbasierter Beweiser für Theoreme der Prädikationlogik verwendet wird.

Das Programmeigenschaften sowohl logische als auch arithmetische Operationen über unbeschränkte Datenstrukturen wie Arrays involvieren, sind für das Erzeugen und Beweisen von Programmeigenschaften effiziente Methoden für das Schlussfolgern in das Schlussfolgern in den Theorien und über den Quantoren notwendig. Ein weiterer Beitrag dieser Arbeit ist die Erweiterung des prädikatenlogikbasierten Theorembeweisers Vampire um Schrankenausbreitungsmethoden zur Lösung von Systemen linearer

Ungleichungen. Somit kann Vampire als automatisches Tool zum Entscheiden der Erfüllbarkeit von Ungleichungssytemen auf der Menge der rationalen Zahlen oder reellen Zahlen verwendet werden. Wir zeigen experimentell, dass die Effizient der Schrankenausbreitung in Vampire im Vergleich mit state-of-the-art Satisfiability Modulo Theory Lösern auf schweren linearen Optimierungsproblemen gut abschneidet.

Unser arithmetischer Löser ist auf Konjunktionen linearer Ungleichungen limitiert. Jedoch haben arithmetische Programmeigenschaften üblicherweise eine komplexere boolesche Struktur aus Kombinationen von Konjunktionen, Disjunktionen und Negierungen. Um daher unsere Arbeit auf komplexe arithmetische Eigenschaften anwendbar zu machen, wurden booleschen Entscheidungsproblem (SAT) Löser in Vampire integriert. Unsere Arbeit nutzt das kürzlich entwickelte AVATAR Framework um das Schlussfolgerungen über Theoreme der Prädikatenlogik von der booleschen Struktur der Probleme zu trennen. Wir beschreiben die technischen und implementierungsbezogenen Herausforderungen die bei der Integration der effizientesten SAT Lösern in Vampire auftreten und verwenden unsere Implementierung zur Evaluierung des AVATAR Frameworks mit einer Vielzahl von Problemen der TPTP Bibliothek automatisierter Theorembeweiser.

Contents

1 Introduction **1**
1.1.1 Problem Statement 4
1.1.2 Thesis Contribution 8

2 Theoretical Preliminaries **11**
2.1 Programing Model 11
2.1.1 Program and Variables 12
2.1.2 Expressions and Semantics 13
2.1.3 Loop Body and Guarded Assignments 15
2.1.4 Loop Invariants 16
2.2 First-order Theorem Proving 17
2.2.1 Inference Systems 18
2.2.2 Saturation Procedure 21
2.2.3 Redundancy Elimination 23
2.3 SAT Solving 25
2.3.1 The DPLL Procedure 26
2.3.2 The Conflict Driven Clause Learning procedure (CDCL) 28

3 First-order Theorem Proving for Program Analysis **33**
3.1 Symbol Elimination for Invariant Generation 33
3.1.1 Update Predicates 34
3.1.2 Loop Properties 35
3.1.3 Reasoning with Theories 39
3.1.4 Invariant Generation 39
3.2 Lingva: Loop Invariant Generation using Vampire 41
3.2.1 Preprocessing 42
3.2.2 Program Analysis 44
3.2.3 Symbol Elimination 47
3.2.4 Post Processing 48
3.3 Experiments using Lingva 51
3.3.1 Academic Benchmarks 52

3.3.2 Open Source Benchmarks . . . 55
3.3.3 Initial Version and Limitations . . . 55

4 Bound Propagation for Arithmetic Reasoning 57
4.1 Bound Propagation Method . . . 57
4.1.1 General Presentation . . . 58
4.1.2 Resolution with Inequalities . . . 59
4.1.3 An Example of Bound Propagation . . . 62
4.2 Integrating Bound Propagation in Vampire . . . 63
4.2.1 Input Problems . . . 64
4.2.2 Representation of Reals . . . 65
4.2.3 Arithmetic in Vampire . . . 66
4.3 Strategies for Variable and Value Selection . . . 69
4.3.1 Variable Selection . . . 69
4.3.2 Assigning Values to Variables . . . 70
4.3.3 Using Vampire's BPA . . . 73
4.4 Experiments using Vampire's new Decision Procedure . . . 74

5 Use of SAT Solvers in Vampire 81
5.1 AVATAR Architecture . . . 81
5.1.1 Setup for AVATAR . . . 82
5.1.2 AVATAR Algorithm . . . 85
5.2 Integrating Lingeling SAT Solver in AVATAR . . . 88
5.2.1 Integrating Lingeling . . . 89
5.2.2 Using Lingeling in Vampire . . . 91
5.3 Experimenting with Vampire's AVATAR . . . 92
5.3.1 Benchmarks and Results . . . 92
5.3.2 Analysis of Experimental Results . . . 96

6 Related Work 99
6.1 Invariant Generation . . . 99
6.2 Arithmetic Reasoning . . . 101
6.3 Reasoning with Theories . . . 103

7 Conclusion 107

Bibliography 111

CHAPTER 1

Introduction

Analyzing and verifying computer programs with million lines of code is an important and challenging task. Banks, hospitals, companies, organizations and individuals heavily depend on very complex computer systems, such as Internet, networking, online payment systems, and autonomous devices. These systems are integrated in an even more complicated environment, using various computer devices. Technically, software systems rely on software implementing complicated arithmetic and logical operations over the computer memory. If this software is not reliable, the costs to the economy and society can be huge. Software development practices therefore need rigorous methods ensuring that the program behaves as expected.

Formal verification provides a methodology for making reliable and robust systems, by proving that programs have no errors and thus are correct. During the past decades, formal verification has gained significant academic and industrial interest. For example, the formal verification of Microsoft drivers reduced the main source of the Windows operating system crashes down to almost zero. Another example is the use of formal verification techniques at Airbus in order to ensure that the software operating Airbus planes respond in a timely manner, in all possible scenarios.

There are various approaches and attempts for formally verifying software systems, for example *automated testing* and *static program analysis*. Testing in general is based on first creating a set of tests cases on which the program runs and then use these test cases in conjunction with the expected output of the program. When executing the program, one is required to ensure that the program behaves correctly on the considered test cases and returns the expected result. The major drawback of test-based verification, [19, 43, 46, 86] is that it does not ensure correctness of the program; it only ensures that the program runs correctly on the tested input. In order to ensure correctness one would need to create a set of tests that cover the entire set of possible input-output combinations of the program, making it practically unfeasible and commercially not

viable. Nevertheless, testing remains very useful during software development. One of the major benefits of test-based software development is that the software is tested during the development process and if bugs occur they are promptly fixed. By doing so, one is keeping the cost for software development low and alongside provides a set of test benchmarks for the refined program.

Contrarily to testing, static software analysis provides efficient and sound approaches proving that the software is correct on all possible inputs. One such a method is *counterexample-guided abstraction refinement (CEGAR)* [23] in model checking [22, 96]. CEGAR works by first abstracting the program and trying to find an unfeasible trace in the abstraction. If such a trace is found and by following the trace in the original program the error state is reached, an error in the program is found. By applying CEGAR, one can hence detect error states in the program and fix them automatically. On the other hand, if the detected unfeasible trace is spurious, that is it cannot happen in the actual program, the program abstraction is refined and used further.

One challenge in static program analysis comes with the treatment of the logically complex part of the code, such as loops and recursion. For such program parts, formal verification requires additional program properties to hold at intermediate points of the program and using these properties to prove that programs have no errors. Typically, such properties are loop invariants summarizing the loop behavior and ranking functions ensuring that the loop is terminating. Loop invariants hold before and after every loop execution and, in most of the cases, they express inductive properties of the program. Providing such properties manually requires a considerable amount of work by highly skilled personnel and makes verification prohibitively expensive. Formal verification therefore requires non-trivial automation and automatic generation of valid program properties, such as loop invariants, is a key step to such automation.

Extracting and generating invariants for programs containing loops is an active research area. Various automated methods based on Craig interpolation [104], abstract interpretation [27, 28] and first-order theorem proving [91] have been proposed during the past years. These methods infer quantified invariants over unbounded data structures, such as arrays, and differ in the employed user guidance. In the case of interpolation, invariants are constructed from interpolants extracted from correctness proofs of the program and they express valid program properties over various data structures used in the program [79]. The generated invariants are universally quantified over array elements and a user-specified postcondition is used in the process of interpolation. User guidance is also required in [51], where the shape of the invariants is specified by user-defined templates. The inductiveness property of the invariants is then used to derive constraints over the unknown parameters of the invariant template. These constraints are further solved using a satisfiability modulo theory (SMT) solver, and an invariant without quantifier is obtained. A different approach is described in [50], where predicate abstraction over an a priori defined set of predicates is used to derive quantified

invariants as the strongest boolean combination of the given set of predicates. Universally quantified invariants over arrays are also inferred in [26, 52], however in a fully automated way, by applying abstract interpretation over array segments. User guidance is also not required in [71], where a first-order theorem prover is used to generate quantified invariants. The distinctive feature of [71] comes with a new method, called symbol elimination, and is the first ever method generating invariants with quantifier alternations.

Symbol elimination works as follows. Suppose we are given a loop L over scalar and array variables. Symbol elimination first extends the loop language L to a richer language L' by additional function and predicate symbols, such as loop counters or predicates expressing update properties of arrays at different loop iterations. Next, we derive a set P of first-order loop properties expressed in L'. The derived properties hold at any loop iteration, however they contain symbols that are not in L and hence cannot yet be used as loop invariants. Therefore, in the next step of symbol elimination, logical consequences of P are derived by eliminating the symbols from $L' \setminus L$ using first-order theorem proving. As a result, first-order loop invariants in L are inferred as being logical consequences of P. In this thesis we exploit symbol elimination in first-order theorem proving, provide a fully automated support for invariant generation using symbol elimination and experimentally investigate to which extend the generated invariants express the "intended" meaning of the program.

There are several challenges for making symbol elimination in first-order theorem proving practically useful. Modern first-order theorem provers, e.g. [63, 87, 94], lack several features essential for implementing symbol elimination for invariant generation. These include *reasoning with first-order theories*, in particular in the combination of the first-order theories of arrays and linear arithmetic, since almost all essential properties about computer memory quantify over the memory content, e.g. array elements, and implement operations over integers. Program properties are however typically expressed as boolean combinations of atomic predicates over data structures, e.g. using logical disjunction, negation and conjunction. Therefore, to use first-order theorem proving for invariant generation, one needs to extend the prover with an *efficient reasoning engine for boolean logic* which can easily be integrated with the theory reasoning part of the prover. Finally, *automated tool support for symbol elimination* in first-order theorem proving is needed. Only this way, software engineers and developers will be able to use invariant generation and symbol elimination results in their work, without the need to become experts in first-order theorem proving. In this thesis we address these challenges and provide solutions as follows:

1. We provide a fully automated tool support, called Lingva, for program analysis and invariant generation using symbol elimination (Chapter 3).

2. We integrate first-order theorem proving with a novel procedure for linear arith-

metic, called bound propagation [67], and experimentally evaluate our integration on a large collection of benchmarks (Chapter 4).

3. We combine first-order theorem proving with satisfiability (SAT) solvers, supporting hence efficient reasoning with both quantifiers and boolean logic (Chapter 5).

1.1.1 Problem Statement

In this thesis we study program analysis and verification using first-order theorem proving. We provide tool support for invariant generation using symbol elimination and implement new procedures for automated reasoning in full first-order theories. In particular, we consider the theories of arrays and linear arithmetic and provide experimental evidence for using first-order theorem proving for program analysis.

The research topics addressed in this thesis are as follows:

Invariant Generation We study invariant generation by symbol elimination in first-order theorem proving. We are interested in the performance of symbol elimination, in particular how efficient can symbol elimination generate useful invariants. In addition to performance, we also investigate the quality of the generated invariants. To this end, we analyze whether the generated invariants imply the "intended" meaning of the program, where the intended meaning of the program is specified by user-given properties.

Theory Reasoning Almost all problems in program analysis and verification require automated reasoning in the combination of theories of various data structures. First-order theorem provers are efficient in handling quantifiers but are weak in theory reasoning. We therefore study how different decision procedures can improve the performance of first-order theorem provers when it comes to generate and prove properties with both theories and quantifiers. When it comes to arithmetic reasoning, well-known methods such as the Fourier-Motzkin method [93] or the Simplex [20] method can be used for solving a system of linear inequalities. While these methods are very efficient and typically used by state-of-the-art satisfiability modulo theory (SMT) solvers, e.g. [11,33,40], they cannot yet be used to reason about both quantifiers and theories. Recently, several new methods, such as GDPLL [80], conflict resolution [64], and bound propagation [67], have been proposed with the purpose of closing the gap between reasoning about the logical and arithmetic structure of the problem. These methods are similar in structure with the DPLL procedure [82] used by SMT solvers. While these techniques cannot yet handle quantifiers, we believe that integrating them with first-order prover can be done without major changes on the underlining reasoning mechanism of the prover. Therefore, in this thesis we integrate and evaluate bound propaga-

tion in the context of a first-order theorem prover for solving systems of linear inequalities.

SAT Solving We experimentally analyze a new architecture for integrating first-order theorem proving with SAT solving. The architecture is based on formula exchange between the first-order reasoning part and the SAT solver. We therefore integrate SAT solving in first-order theorem proving and evaluate how the performance of a first-order theorem prover is changed by using such a combination for proving formulas.

In what follows, we overview in more detail each of the above listed research directions.

Invariant Generation. In this thesis we present a new tool generating quantified invariants without any user guidance. Our tool, called Lingva, uses symbol elimination in first-order theorem proving and infers quantified invariants over arrays, possibly with quantifier alternations, in a fully automated way. Lingva makes use of the Vampire theorem prover [72] and stands for Loop invariant generation in Vampire.

Inputs to Lingva are C programs with (multiple) loops. For each input program loop, Lingva combines program analysis with invariant generation, as follows.

First, the program is analyzed and translated into a collection of logical formulas. To this end, we used Clang [1] as the front-end C parser and construct the abstract syntax tree (AST) of the program. Next the AST is traversed and the semantics of the program is translated into a set of logical formulas, expressed in the internal format of the Vampire theorem prover. For translating programs into logical formulas, we however faced some challenges related to reasoning about different states and program locations. We overcame these challenges by extending the language of the theorem prover with new constructs specific to programming languages. These constructs include *if-then-else* and *let-in* expressions and they allow us to express the transition relation of the program using first-order formulas extended with *if-then-else* and *let-in*. We call first-order formulas with *if-then-else* and *let-in* expressions extended first-order formulas. Moreover, as the considered programs involved arrays and integers, we extended the first-order theorem prover with built-in support for arrays. With such extensions, the next step of program analysis is the collection of valid program properties, expressed as extended first-order formulas. For doing so, Lingva implements various steps of program analysis, based on [70], as follows. First, program variables are collected and classified into updated or constant variables. Among the updated variables, so-called counter variables are identified. A counter variable is a variable that is incremented or decremented by a constant value through the loop. Note that array indexes are typically counter variables. Properties over counter variables are derived by analyzing how the values of these variables are changing throughout the loop. Such properties include (strictly) increasing or

decreasing monotonicity properties, density properties describing at which loop iteration was the variable updated, and the combination of the previous properties. Next, the array variables of the loop are analyzed and so-called update predicates of array variables are inferred. An update predicate of an array expresses at which loop iteration and position was the array updated by what value. Finally, the transition relation of the program is expressed as an extended first-order formula.

By performing program analysis within Lingva, the input program is translated into a set of extended first-order formulas, each formulas describing a valid loop property. However, these properties cannot yet be used as invariants as they use additional symbols not present in the loop, such as the loop counter and update predicates of arrays. Therefore, the next step of Lingva is symbol elimination, that is eliminating the additional symbols from the constructed set of extended first-order formulas. For doing so, Lingva uses symbol elimination in the Vampire theorem prover and derives logical consequences of the extended first-order formulas. The derived logical consequences are such that they only contain symbols present in the loop, and hence they are loop invariants. For implementing consequence generation and symbol elimination using the Vampire theorem prover, Lingva relies on special term orderings used by Vampire and specifies which symbols needs to be eliminated by making these symbols the largest in the term ordering of the prover.

As a result of invariant generation, a large collection of invariants is generated, where some invariants imply each other. Therefore, in the last step of Lingva a minimal set of invariants is inferred, by eliminating invariants that are redundant, e.g. implied by other invariants.

While powerful and generic, symbol elimination in Lingva has various limitations. Lingva only supports loops with nested conditionals, but nested loops are not yet handled. Further, program analysis and invariant generation is only supported for integers and arrays, other unbounded data structures, such as lists or heaps, are not yet supported. The invariant generation and minimization step of Lingva crucially depends on efficient reasoning engines for generating and proving properties with both theories and quantifiers, in particular when it comes to the theory of linear arithmetic. We tried to improve the quality of generated invariants by Lingva, by integrating a new decision procedure, called bound propagation, in the Vampire theorem prover, as described below.

Theory Reasoning. In this thesis we describe how the bound propagation method [67] for deciding satisfiability of a set of linear inequalities can efficiently be integrated in a first-order theorem prover, in particular in Vampire. In addition to deciding satisfiability, we also describe how to construct a model for the input problem, if the problem is satisfiable.

The main steps of bound propagation are similar to the DPLL method [82]. For example, the notion of a clause in the DPLL has its equivalent as a linear inequality in

bound propagation. Similarly, unit clauses in DPLL correspond to value bounds over variables in bound propagation, whereas DPLL's unit propagation is similar to the bound propagation procedure.

In a nutshell, the bound propagation algorithm, denoted in the sequel by BPA, works as follows. Given a system of linear inequalities over reals or rationals, BPA tries to iteratively assign values to variables. After the assignment is done the values are used in order to derive new value bounds on other variables from the initial problem. Derivation of new bounds on other problem variables is called *bound propagation*. The process of bound propagation either derives an inconsistent pair of bounds on some variable, or it solves the system. If inconsistency occurs, then a new inequality is computed. This newly computed inequality is called a *collapsing inequality* and is used to derive a new bound on a previously assigned variable. By doing this step we exclude the previously assigned value to this variable. There are some cases where the newly derived bound is inconsistent with a previously assigned bound. If this happens it means that the original system is unsatisfiable. Otherwise, BPA *backjumps* to the point where a value for the (conflicting) variable was selected, and selects a new value for the variable.

As reported in this thesis, we implemented the BPA decision procedure for solving systems of linear inequalities in Vampire. Let us note some of the challenges that we encountered while implementing BPA. When one tries implementing an efficient version of BPA one has to take into consideration various options that can effect the overall performance of the algorithm. Consider for example the case when a value for a variable is selected. Since we are dealing with rationals or reals we have an infinite domain for the variables' values, hence selecting the right value becomes highly non-trivial. Another distinguishing feature of BPA is the used variable ordering. Unlike other decision procedures, such as like Simplex [20], Fourier-Motzkin [29], GDPLL [80] and conflict resolution [64], bound propagation does not require a predefined variable ordering. This feature gives a lot of freedom for using and experimenting with different variable orderings in BPA. Inspired by SAT/SMT solving, in our work we implemented a couple of strategies for variable and value selection in BPA.

We have evaluated BPA on a large number of examples taken from the SMT community as well as hard linear optimization problems taken from the MIPLIB library [62]. Our experiments are promising and encouraging. For example, there are some hard optimization problems on which BPA outperforms the best existing SMT solvers. Nevertheless, our work has some limitations on the shape of the systems of linear inequalities that are accepted as input. Currently, we can only reason about conjunctions of inequalities, and hence we cannot yet handle an arbitrary boolean structure of inequalities. Similarly to SMT solving, in particular to DPLL(T) [45], such a limitation can be handled by using a SAT solver to reason about the boolean structure of the problem and use BPA for solving a conjunction of linear inequalities. For this reason, in our work we decided to extend BPA, and Vampire in general, with a SAT solver, as detailed below.

SAT solving in First-Order Theorem Proving. Recently a new architecture, called AVATAR, for automated first-order theorem provers has been introduced [103]. The AVATAR architecture combines both first-order reasoning with SAT solvers in order to boost performance of first-order theorem provers on problems with both theories and quantifiers.

In a nutshell, AVATAR works as follows. Given a first-order problem one tries solving it by using a first-order saturation algorithm. In the context of AVATAR the saturation algorithm has to be modified in such a way that the first-order reasoning part interacts with a SAT solver. Inside this framework the SAT solver has the same functionality as a regular incremental SAT solver. The major difference appears on the first-order reasoning part. In this case the job of splitting clauses is delegated to the SAT solver. For achieving this goal one has to create a mapping from the first-order clauses to the SAT clauses and then ask the SAT solver to generate a model. If a model is found this acts as a guide on how the components have to be used in the first-order reasoning part. In case the boolean problem proves to be unsatisfiable, it implies that also the first-order problem is unsatisfiable. Besides these modifications to the saturation algorithm, the AVATAR architecture works with assertions. Therefore, further modifications on how clauses are stored have to be done. Similarly the simplification and deletion rules that are used in the saturation algorithm [83] have to be adapted to the new architecture.

In the case of the Vampire theorem prover, the AVATAR framework was integrated and a default SAT solver was created. After the first experiments reported in [103], it became evident that withing this new framework first-order provers can solve problems that no other solver could solve before. Based on this initial results, in this thesis we study the AVATAR framework in the case of the Vampire first-order automatic theorem prover and integrate Vampire with various SAT solvers, including the best existing SAT solver, Lingeling [17]. We report on our implementation integrating Lingeling within Vampire, and describe our experiments for using the AVATAR framework of Vampire with Lingeling and other, less efficient SAT solvers. Our experiments show that, as expected, using the best SAT solver as a background solver of AVATAR makes Vampire proving most of the problems, including some very hard problems that could not be proved before.

1.1.2 Thesis Contribution

The main focus of the present thesis was to experimentally study and better understand how first-order theorem provers can be used in the context of program analysis and verification. Summarizing, the present thesis brings the following contributions.

1. **Invariant Generation.** We present an efficient implementation, called Lingva, of the symbol elimination method for quantified invariant generation. Our implementation offers fully automated support for program analysis, invariant genera-

tion and minimizing the set of generated invariants. For doing so, in our work we used new extensions of the Vampire theorem prover, such as *if-then-else* and *let-in* constructs, which make first-order provers better suited for program analysis and verification. We also studied the quality of the generated invariants by using our invariants in proving the intended properties for programs. Our experiments show that symbol elimination can generate quantified properties with quantifier alternations that no other solver could derive so far. Our results have been published as a peer-reviewed conference paper in the proceedings of the "Perspectives of Systems Informatics - 9th International Andrei Ershov Memorial Conference (PSI), 2014" [38].

2. **Theory Reasoning.** We extended the first-order theorem prover Vampire with support for arithmetic reasoning. To this end, we implemented the bound propagation algorithm for solving systems of linear real and rational inequalities. Our implementation integrates a wide range of strategies for variable selection and value selection, allowing us to experiment with the best options within bound propagation for solving linear inequalities. Our work was the first ever implementation of bound propagation in a first-order theorem prover. Our results have been published as a peer-reviewed conference paper in the proceedings of the "15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing(SYNASC), 2013" [39].

3. **SAT solving.** Following the new AVATAR architecture for first-order theorem prover, we integrated the Vampire theorem prover with various SAT solvers. In particular, we integrated Vampire with the best performing SAT solver, Lingeling, and evaluated our implementation on a large number of examples. Our results have been published as a peer-reviewed conference paper in the proceedings of the "13th Mexican International Conference on Artificial Intelligence (MICAI), 2014" [18].

The rest of this thesis is structured as follows. In Chapter 2 we make a short overview of the basic notions used throughout this thesis. We continue by presenting the method of symbol elimination for invariant generation in Chapter 3 and describe our implementation and experiments. Next, we present the bound propagation algorithm and its first implementation in Vampire in Chapter 4. Further, we describe Vampire's AVATAR architecture and how we managed to integrate SAT solvers in Vampire in Chapter 5. Chapter 6 overviews related work and Chapter 7 concludes the thesis.

CHAPTER 2

Theoretical Preliminaries

In this section we make an overview of different methods that are relevant for the thesis. We start by first fixing the programming model and describe loop invariants in Section 2.1. The approach for invariant generation used in this thesis is based on first-order theorem proving. In Section 2.2 we next overview the key ingredients of first-order theorem proving and automated reasoning. Recent approaches proposes SAT solvers to be used in first-order theorem proving. Section 2.3 therefore overviews the notions and techniques used in state-of-the-art SAT solvers.

2.1 Programing Model

Automatic discovery of quantified invariants for programs with loops is very important in the framework of program verification and more precisely in static analysis. In the following we are going to present the main ingredients used in our work for generating such invariants.

Our method deployed for automatically generating loop invariants can be viewed in a nutshell as a three stage process. First stage, starting with a loop containing arrays, we first try to extract different first-order program properties (formulas) from it using auxiliary symbols, such as loop counters. In the second stage of this process, using the collected properties (formulas) we then derive the so-called *update predicates* for array variables that appear in the original loop. Finally, in last stage of the process, after collecting all the properties, we run a saturation-based theorem prover to eliminate all the auxiliary symbols and obtain loop invariants expressed as first-order formulas using only symbols that occur in the loop language.

In what follows, we fix the notions that are going to be used throughout the rest of this thesis. The notations and terminology used in this thesis is based on [71].

2.1.1 Program and Variables

Consider a program *P* that contains a single loop and whose body contains assignments, conditionals and sequencing. In the sequel we consider that *P* is fixed and all the given definitions are relativized to it. We denote by *Var* the set of all variables that occur in *P* and by *Arr* the set of all array variables that occur in it.

The *vocabulary (V)* of our program model consists of a countable set of typed variables. The variables that constitute the vocabulary range over data domains used in the actual program, such as booleans, integers or arrays. Other variables are also used in order to express the progress in the execution of the program. Each variable from this vocabulary has a *type* and this indicates the *domain* of that variable (e.g. data variable could range over the natural numbers, while progress variables could range over a finite set of states).

Whenever we speak about programs containing loops we make the assumption that it contains *array variables*, denoted by aa, bb, cc,..., and *scalar variables*, denoted by a,b,c, ... We also introduce a *loop counter* and denoted it by *k*.

Programs Semantics

Semantics of a program containing sequencing, assignments and conditionals is defined in the standard way, as presented by [76]. A simple statement is the basic computation step and is intended to be executed in a single step. In the following we are going to give a brief overview of the statements and constructs that can appear in our programming model.

Assignment statement. Intuitively it can be defined as follows: for a list of variables $\bar{v} \in Var$ and $\bar{e}$ a list of expression of the same length and corresponding type, $\bar{v} := \bar{e}$ is an *assignment* statement.

Sequential composition. The *sequential composition* of two statements S_1 and S_2, denoted by $S_1; S_2$ can be viewed as follows: first the statement S_1 is executed and when it terminates the statement S_2 gets executed. It is easy to see how one can extend sequential composition to more than two statements, $S_1; S_2; \ldots; S_n$ where $n \geq 2$.

Conditionals (if-then-else). We denote *conditional statements* by **if** *cond* **then** S_1 **else** S_2. In this construct, S_1 and S_2 are statements and *cond* is a boolean expression (condition). The intended meaning of such a statement can be described as follows: first the boolean expression *cond* is evaluated. If *cond* evaluates to logical `TRUE` then we execute statement S_1, otherwise, when *cond* evaluates to logical `FALSE` statement S_2 is evaluated. In our model there is also a special case of a conditional statement, that is **if** *cond* **then** S_1 which behaves similar to the if-then-else construct but it executes no instructions if the condition evaluates to `FALSE`.

Loop (while-do). We denote a *loop statement* by **while** *cond* **do** S, where *cond* is a boolean expression and S is called the loop body. We defined the loop body as being the composition of a sequence of statements. The intended meaning of the loop can be described as follows: in the first step the condition *cond* gets evaluated. If it evaluates to `TRUE` then the loop body S gets executed. After S terminates execution the condition gets evaluated again. If the condition evaluates to `FALSE` then the loop terminates. Otherwise the loop body gets executed once more.

A program *P* is a finite sequence of the above statements and is a mapping from program states to states (see the definition of states below). A computation of a program *P* can be viewed as a sequence of states.

2.1.2 Expressions and Semantics

In order to formally describe our invariant generation algorithm we will use the *language of expressions (Expr)*. We make the assumption that the language *Expr* contains constants, including all integer constants, both scalar and array variables (defined as $Var \cup Arr$) and logical variables. Besides them we also assume that $Expr$ contains some interpreted function symbols, like the standard arithmetical function symbols $(+, -, \cdot)$ and interpreted predicate symbols, including the standard arithmetical predicate symbols $\leq, \geq$. We also assume that the expressions are well-typed with respect to a set τ of sorts. In what follows, we denote by i the sort of integers. We define *types* in the following manner: every sort is a type and types can be built from other types using type constructors $\times$ and $\rightarrow$. All the scalar variables that we are using are assumed to have a sort and all the array variables used must have a type $i \rightarrow \tau$. As syntactic sugar instead of writing $aa(e)$ we will write $aa[e]$ to denote the element of array *aa* at position *e*.

For defining the semantics of an expression we make the assumption that every sort has an associated non-empty domain. In the case of i the domain associated to it is the set of integers. We also assume that all the interpreted function and predicate symbols that appear in the language are interpreted by functions and relations of appropriate sorts. For example $\leq, \geq$ are assumed to be interpreted by the standard inequality of integers.

Semantics of the *Expr* language is defined using the notion of *state*. A state σ maps each scalar variable *u* of a sort τ into a value in the domain associated with τ, σ : **Var** $\rightarrow$ **Dom**(τ) and we denote it as $\sigma[u]$. In the case of an array variable of type $i \rightarrow \tau$ into a function from integers to the domain associated with τ. Now given a state σ we can define the value of any expression e over Var in that state inductively as follows.

- The value of a variable $v \in Var$ is simply the value in the particular state, $\sigma[v]$.
- For an expression $f(e_1, e_2, \ldots, e_n)$ we define

$$\sigma[f(e_1, e_2, \ldots, e_n)] = f(\sigma[e_1], \sigma[e_2], \ldots, \sigma[e_n]),$$

  that is the function f is applied to the values of $e_1, e_2, \ldots, e_n$ in the state σ.

The evaluation of formulas is done in the standard way, see e.g. [76]. For simplicity reasons we do not consider arrays as partial functions and we are not analyzing array bounds.

Extended Expressions $e^{(k)}$

In our framework we are dealing only with programs that contain single loops. We assume that the computation of a program P starts in an initial state denoted by σ_0. Then, the state reached by the computation of the loop in the kth loop iteration is denoted by σ_k.

Let us assume that we have a fixed program loop P and some initial state σ_0. The definition of an expression is then parametrized by the initial state σ_0. Also let σ_k be the state obtained after k computations of P starting from σ_0.

Keeping all the previous assumptions in mind, for every integer expression k and a loop variable v of type τ, we denote by $v^{(k)}$ the value of v in the state σ_k. We call $v^{(k)}$ an *extended expression*. The value of the extended expression $v^{(k)}$ is of type τ. A formula ψ, that might contain also extended expressions, is valid if and only if the formula is true for every computation pf P. That is, the formula ψ is true for all computations of P that start in the initial state σ_0.

During the rest of the thesis we denote by $v^{(0)}$ the value of v in the initial state. Also in order to reason about programs and for asserting their properties we will use the notion of extended expression $v^{(k)}$. Note however that extended expressions do not occur in the program.

Relativized Expressions and Formulas

In a similar way as for extended expressions, we consider make expressions and formulas to be relativized to the loop iteration. We denote by $i :: e$ an expression e that is relativized with respect to iteration i and $i :: \psi$ a formula ψ that is relativized to iteration i. In order to define these relativized expressions and formulas, in the sequel we make the assumption that they do not contain any extended expression or sub-expressions. That is these relativized expressions and formulas must not contain any occurrences of terms of the form v^j for some variable v and iteration j.

Definition 2.1.1. *Let i denote a loop iteration. An expression or formula relativized to i is defined inductively, as follow:*

1. *Assuming we have v a loop variable ($v \in Var$) and i an integer expression, then $i :: v \stackrel{\text{def}}{=} v^{(i)}$.*
2. *Given two expressions e_1 and e_2, we have $i :: (e_1[e_2]) \stackrel{\text{def}}{=} (i :: e_1)[i :: e_2]$.*
3. *If e is a constant or a variable (but not an array or scalar variable), then $i :: e \stackrel{\text{def}}{=} e$.*
4. *If f is an interpreted function, then $i :: (f(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} f(i :: e_1, \ldots, i :: e_n)$.*
5. *Given a predicate symbol p, then $i :: (p(e_1, \ldots, e_n)) \stackrel{\text{def}}{=} p(i :: e_1, \ldots, i :: e_n)$.*
6. *Given the formulas $\psi_1, \ldots, \psi_n$ having no occurrences of extended expressions, we have $i :: (\psi_1 \wedge \ldots \wedge \psi_n) \stackrel{\text{def}}{=} i :: \psi_1 \wedge \ldots \wedge i :: \psi_n$. For the other logical connectives we have similar definition.*
7. *Let y be a variable not occurring in i, and ψ a formula having no occurrences of extended expressions. Then $i :: ((\forall y)\psi) \stackrel{\text{def}}{=} (\forall y)(i :: \psi)$.*
8. *Let y be a variable not occurring in i, and ψ a formula having no occurrences of extended expressions. Then $i :: ((\exists y)\psi) \stackrel{\text{def}}{=} (\exists y)(i :: \psi)$.*
9. *Let ψ be a formula and all the variables are occurring in i then we have $i :: ((\forall i)\psi) \stackrel{\text{def}}{=} (\forall i)(\psi)$. Similar for $\exists$.*

2.1.3 Loop Body and Guarded Assignments

We represent the loop body of a program P as a collection of *guarded assignments*. A *guarded assignment* is an expression of the form :

$$G \to \alpha_1; \ldots; \alpha_m \tag{2.1}$$

where G is a formula called the *guard* of the guarded assignment and $\alpha_1; \ldots; \alpha_m$ are assignments. Each of the α's are assignments to either variables or array variables. That is they have the form $v := e$ or $aa[e_1] := e_2$ where $v \in Var$ and $aa \in Arr$ and e, e_1, e_2 are expressions. In this settings the guards of all assignments must be syntactically distinct. That is, the left hand-sides of the guarded assignments are different, syntactically. Further, in the case when for arbitrary i, k such that $i \neq k$, we have α_i of the form $aa[e_1] := e_2$ and α_2 of the form $aa[e_3] := e_4$, then in all states satisfying the guard G the values of e_1 and e_3 are different.

When transforming the program into a collection of guarded assignments we have to ensure that the guards are mutually exclusive and that in every state at least one of the guards is true. In [35, 76] the authors present two different ways of converting a program into a set of guarded assignments. In both works the semantics of a guarded assignment is set to be the one of a simultaneous assignment. That is, given a guarded assignment $G \rightarrow e_1 := e'_1; \ldots ; e_n := e'_n$ than the semantics of this assignment is that of $(e_1; \ldots ; e_n) := (e'_1; \ldots ; e'_n)$, meaning all the expressions are assigned in the same time.

The major problem that arises from transformation of a program into guarded assignments is the fact that the conversion can lead to exponentially many formulas in the size of the program. In order to avoid this problem one has to add more information to the guards. The information that has to be added are usually equalities and inequalities that make the guard hold only on some paths of the program.

Let's assume we have the following sequence of assignments $aa[a] := c_1;\ aa[b] := c_2$ as the loop body, where c_1, c_2 are two different constants. Then we can write the following guarded assignments:

$$a \neq b \rightarrow aa[a] := c_1; aa[b] := c_2;$$
$$a = b \rightarrow aa[b] := c_2;$$

Example 2.1.1. Take as an example of a loop body the statements from Figure 2.1.

```
if (aa[a] == cc[a]){
  bb[b] = a;
  b = b + 1;
 }
a = a + 1;
```

Figure 2.1: Loop body example

Than we can write the following guarded assignments:

$$aa[a] = cc[a] \rightarrow bb[b] = a; b = b + 1; a = a + 1;$$
$$aa[a] \neq cc[a] \rightarrow a = a + 1;$$

2.1.4 Loop Invariants

In the context of program verification, loop invariants are typically expressed as formulas in first-order logic. They are used in order to prove correctness properties (in general) about loops and by extension in proving different algorithms that are employing proved

correct loops. Informally a loop invariant is a loop property that should be true on the entry of the loop and is guaranteed to be true during the execution of the loop. Hence, the invariant is true before and after each loop iterations. One way to formalize the use of loop invariants for proving loop properties is by using the Floyd-Hoare logic [53].

A Hoare triple is a formula of the form $\{P\}\ S\ \{Q\}$, where P is called the pre-condition, Q is called the post-condition and S is the code that is executed. Each triple describes how the execution of a piece of code changes the state of the computation. Using such a formalization one can prove partial correctness of a program. Intuitively one can read a Hoare triple as follows. Whenever P holds in the state before the execution of S, then Q will hold afterward or S does not terminate. In the latter case, there is no "after", so Q can be any statement at all. Indeed, one can choose Q to be *FALSE* to express that C does not terminate. For proving total correctness of a program, one has to extend the notion of partial correctness with termination. This step is achieved by modifying the while rules in order to prove that the program also terminates. That is, whenever S is executed in a state that satisfies P, the execution of the program terminates and after S terminates Q holds.

The problem of proving partial correctness of a loop reduces to the problem of finding a property Inv. In this context Inv holds before the execution of the loop, that is the precondition implies Inv, during the execution of the loop the property is preserved and after the loop terminates Inv and the loop condition imply the post-condition.

Definition 2.1.2. *Assuming we have a formula represented as a Hoare triple of the form* $\{P\}$ ***while*** $cond$ ***do*** $S\ \{Q\}$*, we define* Inv *as being an invariant if and only if the following three conditions hold.*

1. *Initial condition:* $P \Rightarrow Inv$*;*
2. *Iterative condition:* $\{Inv \wedge cond\}\ S\ \{Inv\}$*;*
3. *Final condition:* $Inv \wedge \neg cond \Rightarrow Q$*.*

In general programs have a large set of invariants, but only some can be used in order to prove program correctness. This thesis focuses on the use of an automated first-order theorem prover in order to generate useful loop invariants. In the following we are going to present the main ingredients of first-order theorem proving that are used in our work.

2.2 First-order Theorem Proving

In the following we are going to consider the first-order logic with equality where we allow all the standard boolean connectors and quantifiers. For a more detailed overview of the notions presented in this chapter we refer to [7, 72, 83]. We assume that our

language contains the logical constants TRUE (denoted by $\top$) and FALSE (denoted by $\bot$). We also denote terms by *l,r,s,t*, possibly primed, *x,y,z* variables and constants by *a,b,c,d,e*. Functions are denoted by *f,g,h* and predicate symbols are denoted by *p,q*.

An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. Equality is denoted as usual by $=$. Any atom that contains the equality symbol is called *an equality*, whereas the negation of an equality formula, that is the formula $\neg(l = r)$, is denoted by $l \neq r$. A *literal* is an atom A or its negation $\neg A$. A literal that is an atom (e.g. A)is called *positive* while literals of the form $\neg A$ are called *negative*. In general a *clause* is defined as being a disjunction of literals, denoted by $L_1 \vee \ldots \vee L_n$ with $n \geq 0$. The empty clause, denoted by $\square$, is a clause that contains no literals, or more formally a clause where $n = 0$. In our settings the empty clause is always considered to have the truth value *false*. If a clause contains a single literal we call it a *unit clause*, while a unit clause that contains the $=$ is called *equality literal*. We denote by A atoms, by L literals, by C, D clauses, and formulas by F, G, R, B, possibly with indices.

Let F be a formula with free variable $\bar{x}$, then $\forall F$ (respectively $\exists F$) denotes the formula $(\forall \bar{x})F$ (respectively, $\exists(\bar{x})F$). A formula is called *closed*, or a *sentence*, if it has no free variables. We call a *symbol* a predicate symbol, function symbol or variable. Notice that variables are not symbols and that equality ($=$) is part of the language thus it's not a symbol. A formula is called *universal* (respectively, *existential*) if it has the form $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$), where F is quantifier-free.

A *theory* is any set of closed formulas. If T is a theory, we write $C_1, \ldots, C_n \vdash_T C$ to denote that the formula $C_1, \ldots, C_n \rightarrow C$ holds in all models of T, this notion is equivalent to *axiomatisable theory* from logic. When we are working with theory T, we call all symbols that occur in T *interpreted* while all the other symbols *uninterpreted*.

We call a *substitution* any expression $theta$ of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n \geq 0$. An application of such a substitution on an expression E (term, literal, atom or clause) is denoted by $E\theta$ and is obtained by simultaneous replacement of each x_i by t_i. An expression containing no variables is called *ground expression*. When we write $E[s]$ we mean an expression with a particular occurrence of a term s. If we use the notion of $E[s]$ and then $E[t]$ the latter means that the expression is obtained by replacing the distinguished occurrence of s by t.

A *unifier* of two expression E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. If two expressions have a unifier then they have the so-called *most general unifier*. There are a couple of algorithms that deal with computation of most general unifier, see [56, 77, 92]

2.2.1 Inference Systems

In order to explain how saturation algorithms work in general one needs the notion of inference systems, orderings and selection functions. An inference is a n-ary relation on formulas where $n \geq 0$. We usually write inferences as follows:

$$\frac{F_1 \quad \dots \quad F_n}{F}$$

.

The formulas $F_1, \dots, F_n$ that appear above the line are called *premises* of the inference rule whereas the formula F that appears below the line is called the conclusion of this inference rule. We call an inference system $\mathbb{I}$ a collection of inference rules. We call an *axiom* an inference that has 0 premises.

We call a *derivation* of an inference system a tree built by sequencing inferences in $\mathbb{I}$. A *proof* of a formula F in the inference system $\mathbb{I}$ is a finite derivation and all the leafs are axioms. In a similar manner we say that a derivation of F is from *assumptions* $F_1, \dots, F_n$ if the derivation is finite and all the leafs of this derivation are either axioms or one of the F_i formulas. In the same context a *refutation* is a derivation of $\bot$. Assuming that we speak about a formula F being derivable from the assumptions, that means there exists a derivation of F from the assumptions.

Superposition Inference System

In order to introduce the superposition inference system we first have to define the *simplification ordering ($\succ$)* on therms. $\succ$ is called simplification ordering if it has the following properties:

1. $\succ$ is well-founded, meaning there is no infinite sequence of terms $t_0, t_1, \dots$ such that $t_0 \succ t_1 \succ \dots$ holds.
2. $\succ$ is monotonic, that is assuming $a \succ b$, then $s[a] \succ s[b]$ for all terms a, b, s
3. $\succ$ is stable under substitutions, if $a \succ b$, then $a\theta \succ b\theta$, where θ is a substitution.
4. $\succ$ has the subterm property, if a is a subterm of b and $a \neq b$ then $a \succ b$.

The idea to take from simplification orderings is that they provide a way of saying which of the expressions are "simpler". More details about orderings can be found in [7, 83]. An example of such a ordering is the so called Knut-Bendix ordering [61] that compares ground terms with regard to the number of symbols that appear in each of them.

Besides the orderings we also need a way to select the literals that we are going to resolve upon. More formally a selection function selects in every non-empty clause a non-empty subset of literals. In general when we speak about selected literals, we underline them, e.g. $\underline{L} \vee V$ that means literal L and possibly other literals are selected in $L \vee V$.

As a general presentation one can see the *superposition inference system* as a family of systems that are all parametrised by simplification ordering and selection functions.

For simplicity we assume that the selection function and the ordering are fixed. In the following we are going to present the superposition inference system, denoted by *Sup*.
Resolution:

$$\frac{\underline{A} \vee C_1 \qquad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta}$$

where θ is the most general unifier (mgu in the sequel) of A and A'.
Factoring:

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta}$$

where θ is the mgu of A and A'.
Superposition:

$$\frac{\underline{l = r} \vee C_1 \qquad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta}$$

where θ is the mgu of l and r, where s is not a variable and $r\theta \not\succeq l\theta$, $L[s]$ is not an equality literal.

$$\frac{\underline{l = r} \vee C_1 \qquad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta}$$

where θ is the mgu of a and c, where c is not a variable and $b\theta \not\succeq a\theta$, $t'\theta \not\succeq t[s]\theta$.

$$\frac{\underline{l = r} \vee C_1 \qquad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta}$$

where θ is the mgu of a and c, where c is not a variable and $b\theta \not\succeq a\theta$, $t'\theta \not\succeq t[s]\theta$.
Equality Resolution:

$$\frac{\underline{s \neq t} \vee C}{(C)\theta}$$

where θ is the mgu of s and t.
Equality Factoring:

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta}$$

where θ is the mgu of s and s', $t\theta \not\succeq s\theta$ and $t'\theta \not\succeq t\theta$.

We call a selection function *well-behaved* if the function selects either all the maximal literals or a negative literal. In this case the inference system is *sound* and *refutationally complete*. That is in case the empty clause ($\square$) can be derived from a set S of formulas using Sup, then S is *unsatisfiable*. And by refutationally complete we mean the fact that if the initial set of formulas S is unsatisfiable, then the empty clause will be eventually derived from S using Sup.

2.2.2 Saturation Procedure

A set of clause S is called *saturated with respect to an inference system* $\mathbb{I}$ if and only if for every inference in $\mathbb{I}$ with premises in S the conclusion of applying this inference is still in S. We call *smallest saturate set containing S* the set of all clause derivable from S. Using these notion of saturated set one can reformulate the completeness theorem as follows:

Theorem 2.2.1. *A set S of clauses is unsatisfiable if and only if the smallest set of clauses containing S and saturated with respect to the inference system (superposition inference system in our case) also contains the empty clause.*

The problem that arises is that this theorem does not provide with a constructive way of searching for the empty clause. For this purpose *saturation algorithms* are proposed. The algorithm's job is to guide the search for the empty clause. The only steps that such an algorithm is allowed to take are based on the inference system. More precisely at every step such an algorithm should select an inference from the inference system and apply it. From this formalization of the notion of saturation algorithm one can notice that is really important to have a good strategy for selecting the inference that is to be applied. Also in order to preserve completeness one has to design a *fair* inference selection, that is every possible inference must be selected at some step of the algorithm. We call a *fair saturation algorithm* a saturation algorithm that implements a fair inference selection.

Assuming one uses the Sup inference system and a fair saturation algorithm one can distinguish three possible scenarios for running it on a set of clauses:

1. After a finite number of steps taken by the saturation algorithm the $\square$ is found. In this case we can decide that the input set of clauses is unsatisfiable.

2. We manage to saturate the set, that is the saturation algorithm terminates without generating $\square$. In this particular case we can conclude that the set is satisfiable.

3. The saturation algorithm runs forever without generating the $\square$, in this case we also say that the initial set of clauses is satisfiable.

Note that the third case cannot be implemented as is presented due to the fact that it would run forever or even more precisely it would run out of resources (time, memory, etc). In general theorem provers implement this case in a bit of a different way as follows:

3' Run the saturation algorithm *until the system runs out of resources* but without generating the $\square$ clause. In this case the system will return unknown, since it is not clear whether the initial set of clause is unsatisfiable or we just ran out of resources.

OTTER

Before explaining how saturation algorithms work we have to fix some notions. We call *kept clauses* the set of clauses that are currently stored in the search space. We call a set of clauses to be *passive* if they are kept by the saturation algorithm and they did not yet take part in inferences. We call *active* those clauses that are kept and that did take part in some inferences. As presented also in 1 saturation algorithms use the notion of *unprocessed* clause, that is the initial set of clauses. In the following we are going to present one simple saturation up to redundancy algorithm.

Named after the theorem prover *Otter*, the saturation algorithm was first presented in [78]. A slightly modified and simplified version of this algorithm is presented in 1. In this case the algorithm accepts as input an initial set of clauses and tries to decide their satisfiability. In order to understand how this algorithm behaves in practice we first have to explain how each of the procedures used in the pseudocode behave.

Algorithm 1 A simplified version of the *Otter* algorithm

```
1:  Input init: initial set of clauses;
2:  Var active, passive, generated: set of clauses;
3:  Var given: clause;
4:  active := ∅;
5:  passive := init;
6:  main loop
7:  while passive ≠ ∅ do
8:      given := select(passive);
9:      passive := remove(given, passive);
10:     active := add(given, active);
11:     generated := computeInferences(current, active);
12:     if provable(generated) then
13:         return satisfiable;
14:     self_simplify(generated);
15:     simplify(generated, active ∪ passive);        ▷ Forward simplification
16:     if provable(generated) then
17:         return satisfiable;
18:     simplify(active, generated)                   ▷ Backward simplification
19:     simplify(passive, generated)                  ▷ Backward simplification
20:     if provable(active ∪ passive) then
21:         return satisfiable;
22:     passive := add(generated, passive);           ▷ Create union of two sets
23: endloop
24: return unsatisfiable
```

add(...), remove(...). These procedures do exactly what their name suggests. *Add* adds either a clause to a set of clauses or a it creates the union of two sets of clauses. While *remove* removes a clause from a set of clauses.

computeInferences(clause, setOfClauses). The procedure computes and returns the set of all clauses that can be generated by applying the inferences implemented in the prover. It does it by applying all possible inferences between the *clause* and the set of active clauses *setOfClauses*.

self_simplify(setOfClauses). Procedure that applies simplification rules based only on the clauses that appear in *setOfClauses*. An example of such a simplification rule is rewriting by unit equality.

simplify(what, using). This procedure tries to simplify the set of clauses *what* using clauses from the set *using*. In general, when simplification is done the clauses that get simplified are moved to the *passive* set of clauses. When we simplify the set of *generated* clauses by the already existing clauses from $active \cup passive$ we talk about *forward simplification*. Whereas when we simplify the clauses from *active* and *passive* by the *generated* we speak about *backward simplification*.

One can notice that in practice implementing the algorithm in this exact way is not going to give the best results.

Besides Otter, in the same family of so called *given clause* algorithms, there are some variations that have better practical applicability. The major difference between these algorithms is in the way they handle passive clauses. More precise, the major difference consists in whether passive clauses are used for simplification steps or not. One of these algorithms, called DISCOUNT [6] differs from the algorithm presented in the sense that it does not allow passive clauses to take part into simplifications.

Another algorithm called Limited Resource (short LRS) [90] is a variation of the Otter algorithm that uses passive clauses in simplifications but instead of using all passive and generated clauses it tries to estimate which clauses have no chance in being selected by the selection function by the time limit and it discards them. A more in-depth comparison of these algorithms can be found in [75, 88, 90]

2.2.3 Redundancy Elimination

Assuming that one does a straightforward implementation of a fair saturation algorithm the product won't be an efficient theorem prover. In principle in order to improve one should rater use the notion of *saturation up to redundancy*. The general idea behind redundancy is as follows: given a set of clause S and a clause $C \in S$, C is *redundant* in S if it is a consequence of those clauses in S that are strictly smaller than C with regard to the simplification ordering $\succ$. Note the fact that the problem of redundancy is

undecidable in this formulation. Though state-of-the-art theorem provers implement redundancy elimination. This is done by recognizing the clauses based on some sufficient conditions.

Tautology deletion. A clause is called a *tautology* if it has the following form $A \vee \neg A \vee C$ or $l = l \vee C$. That is, this clause is true under any interpretation, or more formally the clause is *valid*. The fact that this clauses are valid allows theorem proves to safely eliminate them.

Subsumption deletion. A clause C is called *subsumed* by another clause D if the clause C can be obtained from D using two types of operations. 1) Application of some substitution θ. 2) Adding of none or more literals. In other words subsumption can be defined as follows: if by applying $D\theta$ and adding some literals we can derive C then the clause D subsumes C and C is redundant in the search space and can be removed from the search space.

In order to define the process of saturation up to redundancy one has to define the inference process. We call an inference process a sequence (finite or infinite) of formulas $S_0, S_1, \ldots$ denoted by

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots. \tag{2.2}$$

We call a step in this inference process a pair $S_i \Rightarrow S_{i+1}$.

The concept of redundancy allows one to remove clause to the search space, hence a process that uses this notion can be described by two types of steps:

- Step that adds to the search space a new clause obtained by an inference whose premises are in the search space;

- Step that deletes the redundant clause from the search space.

Using these notions one can extend them to use the implemented inference system as follows. Assuming we have $\mathbb{I}$ an inference system, we call a $\mathbb{I}$-process an inference process that in each of its steps either adds a clause that is the conclusion of an inference the the search space or deletes a redundant clause from the search space.

Using these notions we can now define *saturation up to redundancy* and algorithm that implements an inference process. Of course we still want to define a *fair saturation up to redundancy* so that we can guarantee fairness for every initial set of clause S_0.

Generating, Simplifying Inferences and Deletion Rules

In general when implementing one of the saturation algorithms it is desirable to remove as many redundant clauses as possible. The problem of deciding whether a clause is redundant is undecidable in general. But it is still possible to try in finding "cheap" sufficient conditions for an inference to result in a deletion and try to apply such steps

eagerly. Let S be an inference system, and each of the inferences has the following form

$$\frac{C_1 \quad \ldots \quad C_n}{C}.$$

From this perspective we can classify the inferences as *simplifying* if by adding the conclusion of such an inference C, makes at least one of the premises (C_i) redundant in the search space.

All the inferences that are not simplifying are called *generating*. They are named like this due to the fact that they generate a new clause C in the search space instead of simplifying one.

Most of state-of-the-art theorem provers, including VAMPIRE, implement the following principle: *apply simplifying inferences eagerly and apply generating inferences lazily.* That is, try first to minimize the search space as much as possible and postpone the generation of new clauses in the search space as much as possible. This decision influences the design of saturation algorithms as follows: from time to time provers try to search simplifying inferences with the cost of delaying the generating inferences.

Another important issue that has to be carefully integrated is implementation of different deletion rules. Although simplifying inferences are in place, there is still the possibility of redundant clause to be found in the search space. Hence they should be removed. The most common deletion rules implemented by provers are tautology and subsumption deletion.

In the general framework of first-order automated theorem proving dealing with long clauses prove to quickly deteriorate performance of a prover. For this purpose there are a couple of splitting techniques commonly used in theorem provers. In this thesis we evaluate the performance of a new splitting technique that uses a SAT solver for guiding the split, introduced in the new AVATAR architecture [103]. In the following we will briefly overview different notions that are used when dealing with a SAT solver.

2.3 SAT Solving

In this section we are going to introduce the notions used by the satisfiability solvers, or as found in literature *SAT solvers*. The algorithms can be categorized in two main classes *complete* and *incomplete*. In the following we are addressing only some essential algorithms from the sound and complete class, more details can be found in [30,48,97]. We call an algorithm sound and complete if and only if it is guaranteed to terminate and give a correct decision of *satisfiability/unsatisfiability* of a given input formula. Throughout this section when we are referring to a formula ψ we make the assumption that it is in conjunctive normal form (CNF).

Keep in mind that throughout this section we are working with propositional logic. We call a *proposition* a formula that is in CNF. In this logic we are assumed to have a

finite set of boolean variable denoted by $x_1, x_2, \ldots, x_n$. We call a *literal* a variable x_i or its complement $\neg x_i$. A *clause* is a disjunction of distinct literals and we denote it by ω_i. One can see a clause as the set of literals that appear in the clause, $\{l_1, \ldots, l_m\}$. Note that due to our construction it is not allowed to have both l and $\neg l$ in the same clause ω. Also a clause that contains no literals is called *empty clause* and is considered to be inconsistent. A formula ψ is in conjunctive normal form if it is a conjunction of disjunction of literals. More formally CNF formula should have the following shape: $\psi : \omega_1 \wedge \ldots \wedge \omega_n$ with $\omega_i : l_1 \vee \ldots \vee l_m$. Or we can also consider a formula to be a set of clauses, e.g. $\{\omega_1, \ldots, \omega_n\}$. In case a CNF formula ψ contains no clauses we consider it to be valid while if $\emptyset \in \psi$ the formula is said to be inconsistent.

Besides the previous notions in the context of satisfiability checking we also speak about partial assignments. A *partial assignment* ρ for a formula ψ is a truth assignment to a subset of its literal. We call a *unit clause* a clause that contains a single literal. In a similar manner we call *binary clause* a clause that contains two literals.

2.3.1 The DPLL Procedure

The original idea behind this algorithm was first introduced by Davis and Putnam in [32] but it was only a couple of years later that Davis, Logemann and Loveland (DPLL) [31] came up with the form it is widely used also today. Basically the DPLL procedure tries to prune the search space based on the falsified clauses.

Algorithm 2 sketches the basic idea of the DPLL procedure on CNF formulas. In this procedure one tries to repeatedly assign an unassigned literal l in the input formula and recursively search for a satisfying assignment for the formula using the value assigned to l. Commonly in the SAT literature the choice of such a literal l is called the *branching* step while the assignment of l to `TRUE` or `FALSE` is called the *decision* step. Another important notion is the one of *decision level* which basically keeps track the recursion depth.

Notice that after the run of this algorithm and in case the formula proves to be satisfiable we want to have a satisfying assignment constructed. For that purpose at each stage we keep track of the partial assignment (denoted by ρ in 2) for the variables that appear in the original formula. At each recursion level the algorithm first applies unit propagation on the current formula. Unit propagation is a procedure that takes all the unit clauses that appear in the formula, assigns them to `TRUE`, adds them to the partial assignment and goes trough all the clauses where the unit literal appears and propagates the assigned value. By doing so the initial formula gets simplified and some other variables might get assigned.

Although the algorithm is presented in a recursive way, in practice it is usually implemented in a iterative manner. But in order to become competitive one has to take lots of crucial decisions. Next we are going to iterate over a couple of them.

Algorithm 2 Recursive DPLL, starting with an empty partial assignment

1: $\rho := \emptyset$
2: **DPLL-recursive**(ψ: CNF formula, ρ: partial assignment)
3: $UnitPropagate(\psi, \rho)$; ▷ Apply unit propagation on F and add all derived assignments to ρ
4: **if** ($\Box \in \psi$) **then**
5: **return** *Unsatisfiable*;
6: **if** ($\psi == \emptyset$) **then**
7: Print the assignment ρ
8: **return** *Satisfiable*
9: $l := pickLiteral(\rho)$ ▷ pick a literal that is not assigned in ρ
10: **if** (**DPLL-recursive**($\psi|_l, \rho \cup \{l\}$) $==$ *Satisfiable*) **then**
11: **return** *Satisfiable*;
12: **return DPLL-recursive**($\psi|_{\neg l}, \rho \cup \{\neg l\}$)
13: **end**

Key Features of DPLL

Variable and value selection. The way in which variables are selected tends to vary most from solver to solver. Strategies developed in order to select the next variable that has to be assigned influences a lot the overall performance of a state-of-the-art SAT solver. In the literature one can find strategies as easy as picking a randomly not already assigned variable to more complex ones like MOMS [59] (maximum occurrence in clauses of minimum size), VSIDS [81] (variable state independent decaying sum), DLIS [98] (dynamic largest individual sum).
Backtracking schemes Also called backjumps, these schemes allow the solver to retract all the decisions made from a specific point until the conflict. The main advantage of using such procedures consists in the fact that one does not have to backtrack to decision level 0 and start the search but rather go to some "close" point in the decision trace. Usually when we speak about backjumping there are two main techniques, one is conflict-directed backjump which was introduced in [100] or the more recent method called *fast backjumping* used in most solvers like zChaff [81] and Grasp [98].
Two watched literal scheme This scheme was introduced in zChaff [81]. This technique makes the task of constraint propagation much easier. The main idea behind this data structure is that one keeps track for each active clause that is not FALSE under the partial assumption of only two literals. These literals can be either assigned to TRUE or they are not yet assigned. We know that if we manage to find the empty clause the DPLL process will stop and in case a clause is unit it can be immediately satisfied. Hence it is easy to find in each clause two such literals. In order to maintain the two watched literal scheme one is allowed to perform two operations: 1) Suppose that a literal l is set to

FALSE. We go over the literals in the clause and try to find another one (preferably set to TRUE) in that clause to watch. 2) We go trough all the clauses that became inactive (satisfied) by assigning l to FALSE and we make the negation of that literal as being watched ($\neg l$). This second step basically bumps the priority for positive literals over the negative ones.

Using this two watched literal scheme one can easily test if a clause is satisfied under the current assignment, this being done by checking if at least one of the literals is set to TRUE. Another important improvement comes from the fact that upon backtracking one has to do absolutely nothing about the watches. This is due to the fact that the invariants that characterize the watched literals are preserved. More details about how the two watched literals scheme work can be found in [81]

Restarts It was noticed that if one allows the search to restart from decision level 0 better results can be obtained. The way restarts work is as follows: one keeps all the learned clauses but retracts all the decisions made until now and starts the search from the beginning. Since the learned clauses are only clause that prune the search space if there is a solution then the solver will eventually find it. In practice often restarts proved to be beneficial for the overall performance of different solvers.

2.3.2 The Conflict Driven Clause Learning procedure (CDCL)

This procedure is similar to the procedure deployed by the iterative DPLL procedure but with a couple of enhancements, more details can be found in [97]. Algorithm 3 in fact is a variation to the DPLL procedure. The major difference to DPLL is the call to the *ConflictAnalysis* procedure each time a conflict is encountered and the call to *Backtrack* in case backtracking is required. Other than that the structure is preserved. Another issue that is not presented in this general overview is how restarts are implemented, for more details see [8,47]. Besides restarts state-of-the-art solver also allow learned clause deletion so that they speed up the process.

The procedure that are involved in this procedure can be succinctly described as follows:

UnitPropagate(ψ, ρ). This procedure behaves in the exactly like the one presented in the case of DPLL. The only difference to that is that in case of a conflict it sends a signal to the original algorithm.

pickVariable(ψ, ρ) and *decide*(ψ, ρ). These procedures have the role of picking the next variable to assign and its value. They are similar to the ones presented in DPLL.

allVarsAssigned(ψ, ρ). This procedure is used as a stopping criterion for the search algorithm. In case all variables are assigned, the algorithm terminates and returns *satisfiable* and the assignment ρ.

Algorithm 3 Typical CDCL algorithm

```
1:  ρ := ∅
2:  CDCL(ψ: CNF formula, ρ: partial assignment)
3:  if (UnitPropagate(ψ, ρ)==CONFLICT) then                    ▷ Deduce stage
4:      return Unsatisfiable;
5:  dl := 0                                                    ▷ Set decision level to 0
6:  while (¬allVarsAssigned(ψ, ρ)) do
7:      x := pickVariable(ψ, ρ)                                ▷ Decision
8:      val := decideValue(x)
9:      dl := dl + 1                                           ▷ Increment decision level
10:     ρ := ρ ∪ {(x, val)}              ▷ Add variable and value to partial assignment
11:     if (UnitPropagate(ψ, ρ)==CONFLICT) then                ▷ Deduce stage
12:         blevel := ConflictAnalysis(ψ, ρ)                   ▷ Diagnostics stage
13:         if (blevel < 0) then
14:             return Unsatisfiable;
15:         else
16:             Backtrack(ψ, ρ, blevel)
17:             dl := blevel               ▷ Reset decision level due to backtracking
```

ConflictAnalysis(ψ, ρ). This procedure has the job to create the conflict clause and to learn it. They are a couple of techniques that deal with how one can create and learn conflict clauses from the most recent conflict. Some of these techniques will be presented next.

Backtrack(ψ, ρ). Backtracks the search to the level computed by the *ConflictAnalysis* function.

Conflict Analysis

Each time a conflict is encountered this conflict is analyzed, in the case of our algorithm the *ConflictAnalysis* procedure is called. During the analysis of a conflict one or more conflict clauses are learned. Exception is the case where the conflict happens at decision level 0 meaning that the formula is unsatisfiable. In 2 the procedure returns a backtrack level less than 0 such that in the next conditional the stopping criterion is met and *Unsatisfiable* is returned.

In order to learn a clause the solver is representing the decisions in an implication graph, where the conflict is marked by *k*. The procedure starts by visiting all variables that are assigned at latest decision level, computes all the antecedents of those variables and keeps track of variables that are assigned at decision levels less than the current decision level. This procedure is repeated until the most recently decided variable is

reached. After this step using resolution and at each level, resolution is applied using the variable decided at that specific level as variable to resolve upon, in order to select the variable to intermediate clauses are derived until a fix point is reached. When the fix point is reached the procedure stops and the clause resulted is learned. Notice that in the worst case the resolution step can be applied at most the number of variables that appear in the clause times. Also notice that using this learning scheme and backtracking the procedure is still complete and sound since it implements a variation of the DPLL procedure, see [31, 32].

But modern SAT solvers use a slightly modified version of clause learning which is based on the notion of *unit implication point (UIP)*.

Unit Implication Points (UIP's)

The idea of using the UIP as stopping criterion for the learning procedure was first introduced by GRASP [98].

The basic idea behind UIP's comes from graph theory. In the implication graph a UIP is nothing else than the dominator. We say that a vertex v is dominating another vertex x in a directed graph if every path from x to another vertex k contains v. In our case this can be viewed in the implication graph as follows: vertex v (also the first UIP) dominates vertex x with respect to the conflict k. For finding a UIP in the implication graph there is a linear algorithm (at any decision level where a single literal is assigned that variable is a UIP), hence there is not that much overhead added to the solver. It is easy to extend the procedure to stop at any of the UIP's found in the implication graph.

Another interesting issue proposed by Chaff [81] is the fact that learning should be stopped at the first UIP and one should always backtrack given the information from the learned clause. Backtracking has to be done to the highest decision level of the literals involved in the learned clause. Stopping at first UIP and backtracking as presented above is referred in literature as *first UIP clause learning.*

When developing a SAT solver, naive implementation of these procedures do not lead to great improvements. For this reason state-of-the-art SAT solvers besides implementing all the previously mentioned featured pay great attention is to the way different data-structures are implemented.

Improvements

Besides the different clause learning techniques modern SAT solvers also implement different restarting schemes which prove to influence a lot the results obtained by the solvers, see [97] for an overview of techniques.

Also one important role in the performance of state-of-the-art SAT solvers relies in the so called *lazy data structures*. Most of these data structures try to optimize the way in which clauses can be visited and how propagation can be done. By using this kind of

data-structures one can ensure that not much time is lost during propagation and visiting clauses, for details about these data structures see [16, 42, 81, 97, 106].

Another important role is played by the preprocessing techniques applied on the formula before the search can start, more details about the preprocessing steps can be found in [16, 41, 42].

CHAPTER 3

First-order Theorem Proving for Program Analysis

In this chapter we present the techniques used in order to generate loop invariants for programs using a first-order automatic theorem prover. First we introduce the notion of *update predicates*, needed in order to express the properties that are collected from the input program. Afterwards we present details regarding the properties of interest and how these properties can be automatically extracted from loops. Having the properties extracted from loops we then go trough the method of *symbol elimination* and explain how it works. Finally, we describe *Lingva*, a tool that implements all these features. We conclude by presenting experimental results obtained from running Lingva on a large set of programs.

3.1 Symbol Elimination for Invariant Generation

Given a program loop $\mathcal{P}$ we are interested in generating invariants for this loop. By invariants we mean properties that are true after an arbitrary number of loop iterations, where the number of the loop iterations is bounded by the loop counter n. Note that $n \geq 0$. The value that each of the loop variables have at iteration 0 is called *initial value*.

In order to generate loop invariants, we proceed as follows. We first extract first-order formulas that describe the loop behavior. These formulas express valid loop properties relative to the loop iterations. For this reason we introduce the predicate *iter* that is denoting the range of valid loop iterations, formally defined as:

$$(\forall i)(i \in iter \Leftrightarrow 0 \leq i \wedge i < n). \tag{3.1}$$

Notice that whenever we speak about an iteration we refer to a value in the interval $[0, n-1]$. Besides the iteration predicate for extracting loop properties we make use of extended expressions, as defined in 2.1.2.

3.1.1 Update Predicates

In order to "force" a saturation based first-order theorem prover to generate loop invariants one needs to extract first some properties about the loop and its variables. The program analysis part of the symbol elimination method is based on the analysis of how arrays are updated during the execution of the loop [71]. In order to describe the array behavior, we introduce so-called *update predicates* and some axioms about them, as follows.

The *update predicates* are constructed as follows: for each of the array variables that are updated in a loop we introduce two predicates. Assume we have an array variable V that is updated during the loop. We introduce the following two update predicates:

1. $upd_{aa}(i, p)$: this predicate says that the array aa gets updated at loop iteration $i \in iter$ at position p.

2. $upd_{aa}(i, p, x)$: this predicate says that the array aa gets updated at loop iteration $i \in iter$ at position p with value x.

These predicates can be automatically extracted from guarded assignments that we collect for representing the program.

Example 3.1.1. Take the program from Figure 3.1. In this case only the array bb gets updated during the loop. Hence, we only introduce the updated predicates $upd_{bb}(i, p)$

```
1  int a=b=0;
2  while(a<m){
3    if (aa[a]==cc[a]){
4      bb[b]=a;
5      b = b+1;
6    }
7    a = a+1;
8  }
```

Figure 3.1: Example program

and $upd_{bb}(i, p, x)$, as follows.

We start by writing down the guarded assignments that can be extracted from this simple loop.

$$aa[a] == cc[a] \to bb[b] := a; b := b + 1; a := a + 1; \tag{3.2}$$

$$aa[a] \neq cc[a] \to a := a + 1; \tag{3.3}$$

Using the information from the guarded assignment (3.2) one can automatically generate the following update predicates for array bb:

$$\begin{aligned} upd_{bb}(i, p) \iff & i \in iter \wedge p = b^{(i)} \wedge \\ & aa^{(i)}[a^{(i)}] == cc^{(i)}[a^{(i)}]. \end{aligned} \tag{3.4}$$

$$\begin{aligned} upd_{bb}(i, p, x) \iff & i \in iter \wedge p = b^{(i)} \wedge \\ & aa^{(i)}[a^{(i)}] == cc^{(i)}[a^{(i)}] \wedge \\ & x = cc^{(i)}[a^{(i)}]. \end{aligned} \tag{3.5}$$

3.1.2 Loop Properties

After we have defined and introduced the update predicates for all array variables that are modified during the loop, we proceed to extract first-order properties of the loop, as described below.

Update Properties of Arrays

One valid loop property about arrays states that if an array is never updated at a position *p* during the execution of the loop than its value remains constant at that position, We will refer to this property as *stability property*. Another property expresses the fact that if an array is updated at some iteration *p* and afterwards is never updated, then the value of the array at that position remains the value that the array was updated with at iteration *p*. This property can be expressed using the update predicates and we will call this property *last update* for an array.

Example 3.1.2. Let us consider again the example from Figure 3.1. Let us analyze the array bb. In this case one can write the stability property as in equation (3.6) where n is the loop counter.

$$(\forall i) \neg upd_{bb}(i, p) \Rightarrow bb^{(n)}[p] = bb^{(0)}[p]. \tag{3.6}$$

Using the same settings as before we can formally state the last update property of bb, as in equation (3.7):

$$upd_{bb}(i, p, v) \wedge (\forall j, j > i) \neg upd_{bb}(j, p) \Rightarrow bb^{(n)}[p] = v. \tag{3.7}$$

In a similar manner we would like to state that some array remains constant during the execution of the loop. For doing so, in the case of the array aa from the example of Figure 3.1, one needs to add the property: $(\forall i)(aa^{(i)} = aa^{(0)})$.For our purposes, in the sequel for a constant arras we do not introduced extended expressions but use the array without changes, as it is. Take the case of Figure 3.1, we simply write aa instead of $aa^{(i)}$ and, for example $aa[p]$ instead of $aa^{(i)}[p]$.

Monotonicity Properties of Scalars

Another set of valid loop properties extracted in the process of analyzing the loop describe valid properties of scalar variables. We call a scalar variable v *increasing* if it has the following property: $(\forall i \in iter)(v^{(i+1)} \geq v^{(i)})$, that is at every iteration of the loop the value taken by this variable is increased. Similarly we call a variable v *decreasing* if it has the property $(\forall i \in iter)(v^{(i+1)} \leq v^{(i)})$ for all possible iterations of the loop. Using these notions we call a variable v to be *monotonic* if it is either *increasing* or *decreasing*.

Monotonicity properties about loops can be either discovered by different program analysis tools or by simply applying some light-weight techniques to discover it. In our case we rely on the light-weight techniques. Let us start by giving an example of such a technique. Assuming that during the computation of a loop some variable v gets assigned expressions of the form $v := v + const$ where const is a non-negative constant, then it is safe to conclude that this variable is *increasing*.

Example 3.1.3. Let us take again the program from Figure 3.1. Using the reasoning above, we can safely conclude that variables a and b are *increasing*.

A refinement of the increasing (decreasing) properties is the *strictly increasing* (*strictly decreasing*) property. A variable v has the strictly increasing property if throughout the execution of a loop it preserves the invariant $(\forall i \in iter)(v^{(i+1)} > v^{(i)})$.

Example 3.1.4. In the example of Figure 3.1, both variables a and b are strictly increasing.

Another property of scalars that we extract is based on the so-called *dense variables*. We call an integer variable v to be dense if every value of the variable is visited by the loop in an increasing/decreasing manner. That is, the value of v is increased (decreased)

at every computation of the loop by constants 0 or 1. Formally, this property can be expressed as:

$$(\forall i \in iter)(v^{(i+1)} = v^{(i)} \vee v^{(i+1)} = v^{(i)} + 1).$$

Using the combination of the above properties, we also extract the following loop properties about scalars.

1. Assuming there is a strictly increasing and dense variable v we will add

$$(\forall i)(v^{(i)} = v^{(0)} + i).$$

   Notice that in this formula we are not restricting the computation to range only over iterations. Also notice the use of the initial value of the variable $v^{(0)}$. It is not hard to argue that this is the case, since at every computation step we are incrementing the value of v with 1.

2. In case the variable v is strictly increasing but not dense then we can add

$$(\forall j)(\forall k)(k > j \Rightarrow v^{(k)} > v^{(j)})$$

   to the set of generated properties.

3. In case a variable v is increasing but not strictly increasing we add

$$(\forall j)(\forall k)(k \geq j \Rightarrow v^{(k)} \geq v^{(j)}).$$

4. In case the variable v is increasing and dense but not strictly increasing, then we have to add the following property

$$(\forall j)(\forall k)(k \geq j \Rightarrow v^{(j)} + k \geq v^{(k)} + j).$$

In a similar manner to the one presented above one can derive the rules for decreasing variables.

Example 3.1.5. Let us consider again the example for Figure 3.1. Using the analysis described above, we extract the following properties:

$$\begin{aligned} &(\forall i)(a^{(i)} = a^{(0)} + i) \\ &(\forall j)(\forall k)(k \geq j \Rightarrow b^{(k)} \geq b^{(j)}) \\ &(\forall j)(\forall k)(k \geq j \Rightarrow b^{(j)} + k \geq b^{(k)} + j) \end{aligned} \tag{3.8}$$

Update Properties for Monotonic Variables

Recall the fact that the loop we analyze is represented as a set of guarded assignments. Suppose that v is a monotonic variable. We would like to express the fact that if the variable gets updated (in other words, the value of this variable changes) then there exists an iteration where the conditions in the guards got satisfied. For this reason suppose that the set $\mathcal{U}$ contains all the guarded assignments where the variable v can be changed. Note that in general we write down the guarded assignments as

$$G_1 \to \alpha_1$$
$$\dots$$
$$G_m \to \alpha_m$$

Using this notation one can define $\mathcal{U} = \{1, \dots, m\}$ as the set of guarded assignments that can change the value of x. In other words one can say that $u \in \mathcal{U}$ if and only if α_u contains an assignment to variable v. Suppose that v is increasing, for decreasing case the formulas are similar to the ones presented below, and also dense. Then we add the following property to the set of generated properties:

$$(\forall val)(val \geq v^{(0)} \wedge v^{(i)} > val \Rightarrow (\exists i \in iter)(\bigvee_{u \in \mathcal{U}} (i :: G_u) \wedge v^{(i)} = val)). \tag{3.9}$$

In the case where v is not dense, but still increasing, we have :

$$(\forall val)(val \geq v^{(0)} \wedge v^{(i)} > val \Rightarrow (\exists i \in iter)(\bigvee_{u \in \mathcal{U}} (i :: G_u) \wedge val \geq v^{(i)} \wedge v^{(i+1)} > val)). \tag{3.10}$$

Example 3.1.6. Consider the example of Figure 3.1. Using the analysis above, we obtain the following property:

$$(\forall val)(val \geq b^{(0)} \wedge b^{(i)} > val \Rightarrow (\exists i \in iter)(b^{(i)} = val \wedge aa[a^{(i)}] == cc[a^{(i)}])). \tag{3.11}$$

Translation of Guarded Assignments

Suppose that $G \to e_1 := e_1'; \dots ; e_k := e_k'$ is a guarded assignment in the loop representation and that $v_1, \dots, v_l$ are all scalar variables of the loop not belonging to $\{e_1, \dots, e_k\}$. Define the *translation* $t(e_j)$ *at iteration i* of a left-hand side of an assignment as follows: for a scalar variable x, we have $t(x) \stackrel{\text{def}}{=} x^{(i+1)}$ and for any variable X and expression e we have that $t(X[e]) \stackrel{\text{def}}{=} X^{(i+1)}[e^{(i)}]$. Then we add the following axiom:

$$(\forall i \in iter)(i :: G \Rightarrow \bigwedge_{j=1,\dots,k} t(e_j) = (i :: e_j') \wedge \bigwedge_{j=1,\dots,l} v_j^{(i+1)} = v_j^{i}). \tag{3.12}$$

Example 3.1.7. Consider again the example from Figure 3.1. We then obtain the following formula:

$$(\forall i \in iter)(aa[a^{(i)}] == cc[a^{(i)}] \Rightarrow bb^{(i+1)}[b(i)] = a^{(i)} \wedge \\ b^{(i+1)} = b^{(i)} + 1);$$

3.1.3 Reasoning with Theories

In general first-order theorem provers are really efficient when it comes to reasoning with quantifiers, but they are not that performant when it comes to reasoning with theories. In order to generate loop invariants one however has to reason with different theories, such as the theory of integers, reals, arrays, and lists. In order to overcome the problem of theory reasoning withing first-order theorem proving, a straightforward solution is to add first-order axioms of the respective theories to the theorem prover. The downside of this approach is the fact that there is no complete axiomatization for the previous first-order theories if we take into account arbitrary quantification.

Nonetheless, one may use incomplete but sound axiomatization of theories. This is the approach taken in [71] and the approach we followed in this thesis for invariant generation. For our purposes, we considered the following incomplete axiomatization of linear integer arithmetic to be added to the first-order theorem prover:

$$\begin{aligned}
&x \geq y \Leftrightarrow x > y \vee x = y \\
&x > y \Rightarrow x \neq y \\
&x \geq y \wedge y \geq z \Rightarrow x \geq z \\
&succ(x) > x \\
&x \geq succ(y) \Leftrightarrow x > y
\end{aligned}$$

where $>$ and $\geq$ are the greater and greater equal arithmetical relations over integers, x, y are integer variables and the successor function $succ$, which is equivalent on writing $succ(e) \Leftrightarrow e + 1$.

In the context of program verification and invariant generation reasoning with theories in an first-order theorem prover is a really hard task. By using simple axiomatization like the one presented above proves to behave well in most situations, but it is still a challenge to reason about more complex arithmetic. One attempt for integrating a more powerful technique for arithmetic reasoning is presented in [39, 67] and its implementation is discussed in Chapter 4.

3.1.4 Invariant Generation

In a nutshell our approach of generating invariants can be described as follows. Starting with a loop that we are interested in analyzing, L we first collect all the variables that appear in the loop. Next the language of this loop gets extended to a richer language. After

the loop language is extended, light-weight analysis is applied and properties about the loop are automatically extracted. After this step we write the loop as a set of guarded assignments. At this point we have a collections of loop properties that contain also symbols that are not part of the original loop, e.g. update predicates. In order to generate invariants we need to mark these symbols in order for the saturation theorem prover to eliminate them when it generates invariants. As last step we make use of a saturation theorem prover in order to generate properties (invariants), containing only symbols that appear in the original program, from the set of extracted loop formulas.

Symbol Elimination for Invariant Generation

Until now we discussed how valid loop properties, including extended expressions, are generated. Now having a collection of first-order properties one needs to add them to a theorem prover so that invariants can be generated. Notice that any consequence that is generated from the set of properties is an invariant. But not all of them are useful invariants. An invariant is useful if and only if it uses only symbols that appear in the original loop. In order to "force" a saturation based theorem prover to generate only properties that satisfy we make use of the so called *well-colored derivations* [54, 79].

The idea behind well-colored derivations (also called local proofs or split proofs) is as follows. We define some of the predicates to be colored while others are uncolored. A symbol, literal or formula is called colored if it has a color, otherwise it is called transparent. We call a derivation to be *well-colored* if any inference applied in that derivation uses symbols that have at most one color. We call *symbol eliminating inference* and inference that has at least one colored premise and the conclusion is transparent.

In the context of invariant generation we can formulate the problem by using one color, that is, we mark all the newly introduced symbols (symbols that are not in the loop but used in the loop properties, such as loop counters) as being not useful for the generated invariants. We leave all the symbols that can appear in the invariant, program symbols and theory symbols, as being transparent. We call an *invariant* for the initial program a property that uses only transparent symbols. Starting from the initial set of properties and applying symbol elimination inferences on the problem is guaranteed to obtain a valid transparent formula. Hence the problem of generating invariants can be formulated in terms of elimination of colored symbols.

However, for making a theorem prover efficient with the task of generating transparent consequences one has to change the literal selection and simplification ordering so that colored symbols are greater in the ordering than any other transparent symbol. This way, the theorem prover will pick colored symbols first, applies inferences over literals containing colored symbols and eliminates colored symbols in the process of saturation over the generated set of first-order loop properties. Such orderings can be constructed using minor changes to the standard Knut-Bendix ordering [74] – see e.g. [54].

3.2 Lingva: Loop Invariant Generation using Vampire

We implemented our approach for invariant generation described in Section 3.1. Our implementation provides a fully automated tool support, called Lingva, for generating and proving program properties, in particular loop invariants.

A overview of Lingva's workflow is presented in Fig. 3.2. Lingva makes use of the first-order theorem prove Vampire [72] and is mainly implemented in C++. Besides the core which is implemented in the framework of Vampire, Lingva consists also of a couple of Python scripts that make it more user friendly. One can download Lingva compiled for Linux x64 machine from `www.complang.tuwien.ac.at/ioan/lingva.html`. In the following we present the main ingredients used in order to develop Lingva. For running Lingva one has to execute the following command:

```
./Lingva problem.c [options]
```

where `problem.c` is a C/C++ program with loops and `options` is a set of optional parameters that control the behavior of Lingva. As output to this query Lingva creates a file `problem_annotated.c` which contains the original loop annotated with the generated loop invariants.

When compared to other implementations, see e.g [55], the preprocessing part and the code annotation and conversion parts of post processing are new features. Further, Lingva extends that approach by more sophisticated path analysis methods and built-in support for reasoning in the first-order theory of arrays. These features allow Lingva to handle a programs with multiple loops and nested conditionals and derive quantified invariants that could not yet be obtained by [55], as arrays and integers, and their axiomatization, were not yet supported as built-in theories in [55].

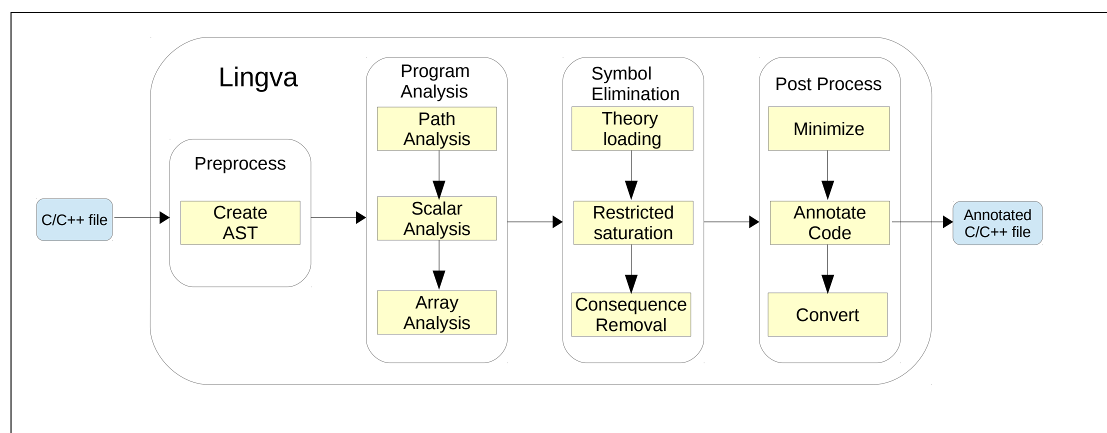

Figure 3.2: The overall workflow of Lingva.

3.2.1 Preprocessing

Input programs of Lingva are first parsed using the Clang/LLVM infrastructure [73] and the abstract syntax tree (AST) of the input is created. Although Lingva front-end can parse arbitrary C/C++ programs, program analysis in the next step has implemented support for a restricted programming model, as follows. We only handle program loops with sequencing, assignments, and nested conditionals. Nested loops, recursive or procedure calls are thus not yet supported. Further, we only treat integers and arrays. Also we restrict program tests and assignments over integers to linear arithmetic expressions. If these restrictions are not met, Lingva terminates with an error message that provides information on the violation of the programming model.

```
void main() {
 //loop 1
 while (condition) {
        loop_body
 }
 //...
 //loop_k: Partition_Init
 int a, b, m;
 int *aa, *bb, *cc;
 while (a < m) {
    if (aa[a] == cc[a]) {
        bb[b] = a;
        b = b + 1;
        }
    a = a + 1;
 }
 //...
}
```

Figure 3.3: Example of a input source code.

Code Transformation

In order to address the problem posed by the `if-then` construct one has to transform it into an `if-then-else`. From a programming point of view one can see the if-then construct as being nothing else but an if-then-else with a `skip` statement on the else branch, see Figure 3.4(a). Or one can ppulltthe first statement after the if on both

branches of the if, see 3.4(b). Assume that we are trying to analyze the program from Figure 3.3 and more precisely we are interested in analyzing loop k. This loop can be safely transformed into the loop presented in Figure 3.4 without changing its meaning.

```
//loop k
while(a<m){
  if (aa[a]==cc[a]){
    bb[b]=a;
    b = b+1;
  }else{
    skip ;
  }
  a = a + 1;
}
```
(a)

```
//loop k
while(a<m){
  if (aa[a]==cc[a]){
    bb[b]=a;
    b = b+1;
    a = a+1;
  }else{
  a = a+1;
  }
}
```
(b)

Figure 3.4: Transformed if-then from Fig 3.3 into an if-then-else construct

Another transformation that is done as soon as the AST gets constructed is conversion of for loops in to while loops. Notice that throughout the presentation of the program analysis methods presented we only use while loops. In general for array manipulation besides while loops also for loops are as used and they appear often in software. In order for us to analyze for loops we convert them into while loops. There are a couple of restrictions imposed on the for loops that can be converted. Recall that a typical for loop looks like :

```
for ( counter ; condition ; increment ) { body; }
```

where usually the `counter` is represented by an integer, `condition` is the stopping condition of the for loop and `increment` represents how the counter gets incremented (notice that it could get decremented as well). We restrict the loops that can be converted to those that use an integer as a counter. The termination condition is a boolean expression, like in the case of while loops and where the increment is explicit in the for construct. The last constrain can be relaxed by simply assuming that the increment is in the body of the for loop. Using this constraints we can convert the classical for loop into a while loop

```
counter; while (condition) { body; increment; }
```

without losing its meaning. The transformation is accomplished by appending the increment (decrement) statement at the for body, use the stopping criteria (`condition`) as

the while stopping condition and initialization of the counter gets done before entering the loop. Such a transformation is nicely illustrated by Figure 3.5

```
for (int a=0; a<m; a++){
  if(aa[a]==cc[a]){
        bb[b] = a;
        b=b+1;
  }
}
```
(a)

```
int a=0;
while(a<m){
  if(aa[a]==cc[a]){
        bb[b] = a;
        b=b+1;
  }
  a=a+1;
}
```
(b)

Figure 3.5: Transformation of for loop (a) into a while loop (b)

After the AST construction and after all transformations are applied, each program loop is analyzed by default by Lingva. However,the user can also specify which loop or set of loops should be analyzed by calling Lingva with the option `-f fn.loopNr`. Where `fn` is the name of the input's C/C++ function block and `loopNr` gives the loop number of interest within `fn`.

Example 3.2.1. Consider Figure 3.3. It is written in C/C++ and contains multiple loops, each loop being annotated with a natural number starting from 1. For simplicity, we only show and describe Lingva on the kth loop of Figure 3.3; analyzing the other loops can be done in a similar manner. The kth loop of Figure 3.3 takes two integer arrays aa and cc and creates an integer array bb such that each element in bb describes an array position at which the elements of aa and cc are equal. This loop is the `Partition_Init` program from Section 3.3. For running Lingva only on this loop, one should execute the command:

```
./Lingva problem.c -f main.k
```

3.2.2 Program Analysis

Program loops are next translated into a collection of first-order properties capturing the program behavior. These properties are formulated using the TPTP syntax [102]. Note that in TPTP, symbols starting with capital letters denote logical variables which are universally (!) or existentially (?) quantified. In order to illustrate how Lingva works and how does the output look we use the TPTP notation.

During program analysis, we extend the loop language with additional function and predicate symbols, as follows. For each loop, we use an extra integer constant $n \geq 0$

denoting the number of loop iterations and introduce an extra predicate $iter(X)$ expressing that the logical variable X is a valid loop iteration, that is $0 \leq X < n$. Loop variables thus become functions of loop iterations, that is a loop variable v becomes the function $v(X)$ such that $iter(X)$ holds and $v(X)$ denotes the value of v at the Xth loop iteration. For each loop variable v, we respectively denote by $v0$ and v its initial and final values. Finally, for each array variable we introduce so-called update predicates describing at which loop iteration and array position the array was updated. For example, for an array bb we write $updbb(X, Y, Z)$ denoting that at loop iteration X the array was updated at position Y by the value Z.

For each loop, we next apply path and (scalar and array) variable analysis in order to collect valid loop properties in the extended loop language. Within path analysis, loops are translated into their guarded assignment representations and the values of program variables are computed using let-in and if-then-else formulas and terms. Unlike [55], the use of let-in formulas (`let...in`) and if-then-else terms (`ite_t`) allow us to easily express the transition relations of programs. These constructs became also standard in the TPTP [102] library recently. Using them one can express the transition relation in a more straight-forward manned than by using guarded assignments.

Example 3.2.2. Suppose that one uses the program from example 3.6. It is straight

```
if ( x > 0 ){
        x = x + 1
}else{
        x = x + y
}
```

Figure 3.6: A simple if-then-else construct

forward to represent this into a combination of let-in and if-then-else constructs. Let us assume that x_1 is the value of x in the next state. Then we can write down the following formula:

$$\begin{aligned} x_1 \;:=\; & if\;(\,x \;>\; 0\,)\;\; then\;(\,let\, x \;=\; x \;+\; 1\; in\; x\,) \\ & \qquad\qquad\quad else\;(\,let\, x \;=\; x \;+\; y\; in\; x\,) \end{aligned}$$

Now going back the *k*th loop of the example 3.3, we can safely add the following translation of guards to the generated problem.

$$\begin{aligned} aa(a) \;==\; cc(a) \;\Rightarrow & a(\,\$successor(X_0)) \;=\; (\,let\, bb(X_1) \;:= \\ & \$ite_t(\,b \;=\; X_1, a, bb(X_1)\,)\; in\; (\,let\, b \;:= \\ & \$sum(b, 1)\; in\; (\,let\, a \;:=\; \$sum(a, 1)\; in\; bb(X_1)))). \end{aligned}$$

Further, (i) we determine the set of scalar and array program variables, (ii) compute monotonicity properties of scalars by relating their values to the increasing number of loop iterations, (iii) classify arrays into constant or updated arrays, and (iv) collect update array properties. As a result, for each program loop a set of valid loop properties is derived in the extended loop language.

```
...
23. ![X2]: bb(X2) = bb('$counter',X2)
22. ![X1]: bb(0,X1) = bb0(X1)
...
16. ![X0, X2, X3]: updbb(X0,X2,X3) => bb(X2) = X3
15. ![X0, X1, X2, X3]: updbb(X0,X2,X3) <=> (let b := b(X0) in
    (let a := a(X0) in (let bb(X1) := bb(X0,X1) in (aa(a) =
    cc(a) & (a = X3 & iter(X0) & b = X2)))))
...
9. ![X0]: iter(X0) => a(X0) = a0 + X0
8. a(0) =a0
...
2. iter(X0) =>(let a := a(X0) in (let b := b(X0) in
   ( let bb(X1) := bb(X0,X1) in a<m )))
1. ![X0, X1]: let a := a(X0) in (let b := b(X0) in
   (let bb(X1) := bb(X0,X1) in (aa(a) = cc(a) =>
   b(X0+1) = ( let bb(X1) := ite_t(b=X1,a,bb(X1)) in
   (let b := b +1 in (let a := a + 1 in b))))))
```

Figure 3.7: Partial result of program analysis

Example 3.2.3. Consider the kth loop of Figure 3.3. A partial set of first-order properties generated by Lingva in the extended loop language is given in Figure 3.7. Properties `1-2` are derived during path analysis. They express the value of the scalar b during the program path exhibiting the then-branch of the conditional within the loop and, respectively, the loop condition. Properties `8-9` are derived during scalar analysis. They state that the values of a are monotonically increasing at every loop iteration; moreover, these values are exactly defined as functions of loop iterations and the initial value $a0$ of a. Properties `15-16` are inferred during array analysis, more precisely when update predicates for arrays are introduced, and they are expressing how the array bb gets updated. Properties `22-23` are inferred during array analysis, and express respectively, the initial and final values of the array bb.

3.2.3 Symbol Elimination

During this step of symbol elimination, for each of the loops that are specified for being analyzed we derive loop invariants. The main ingredient in deriving loop invariants is Vampire [72] framework. The entire work behind symbol elimination is delegated to Vampire so that it will generate logical consequences of the properties that are collected during program analysis steps.

Up to this point by default we start by first loading, in Vampire, the built-in theories of integers and arrays. This was achieved by extending Vampire in order to have build-in support for integer data types. That is now in case one wants to formulate properties using integers there is no need to express them based on the successor function. Rather we simply use the newly added data type. For this purpose some extra predicates and functions that handle integers where added. These predicates are : addition ($\$sum(x, y)$), subtraction ($\$minus(x, y)$ and the special case, unary minus $\$uminus(x)$), multiplication ($\$mul(x, y)$), successor and the standard inequality relations, $<$ ($\$less(x, y)$), $>$ ($\$greater(x, y)$), $\leq$ ($\$lesseq(x, y)$) and $\geq$ ($\$greatereq(x, y)$).

Assuming that one formulates different properties that contain one of the previous built-in predicates and functions and adds them to Vampire, the axiomatization is automatically loaded by Vampire. In case one wants to use it's own axiomatization for different operations, this is also possible. When this happens Vampire's built-in theory axiomatization is not loaded, but rather it uses just the axioms provided by the user.

Recall that Vampire was the first prover that implemented support for let-in and if-then-else expressions. Since the properties collected during program analysis might contain either one of these constructs the first step is to convert the properties into first-order formulas that do not use let-in and if-then-else terms (one can see this step as a preprocessing step). Unlike the initial work from [55], Lingva supports now reasoning in the first-order theories of arrays and uses arrays as built-in data types.

By using the built-in theory axiomatization of arrays and integers arithmetic within first-order theorem proving, Lingva implements theory-specific reasoning and simplification rules which allows to generate logically stronger invariants than [55]. Besides generating invariants, Lingva is used also for proving that some of the generated invariants are redundant (as explained in the post processing step of Lingva).

Next, we collect the additional function and predicate symbols introduced in the program analysis step of Lingva and specify them to be eliminated by the saturation algorithm of Vampire; to this end the approach of [71] is used. As a result, loop invariants are inferred. Symbol elimination within Lingva is run with a 5 seconds default time limit. This time limit was chosen based on our experiments with Lingva: invariants of interests could be generated by Lingva within a 5 seconds time limit in all examples we tried. The user may however specify a different time limit to be used by Lingva for symbol elimination.

```
...
tff(inv3,claim,![X0 :$int]: aa(sk1(X0))=cc(sk1(X0)) |
    ~$less(X0,$sum(b,$uminus(b0))) | ~$lesseq(0,X0)).
...
tff(inv10,claim,![X0:$int,X1:$int,X2:$int]:
    ~(sk1(X0)=X1) | ~($sum(b0,X0)=X2) | ~$less(X0,
    $sum(b,$uminus(b0))) | ~$lesseq(0,X0) | bb(X2)=X1)).
...
```

Figure 3.8: Generated invariants for loop k from Figure 3.3.

Example 3.2.4. The partial result of symbol elimination on Figure 3.7 is given in Figure 3.8. The generated invariants are listed as typed first-order formulas (`tff`) in TPTP. The invariants `inv3` and `inv10` state that at every array position $b0 + X0$ at which the initial array $bb0$ was changed, the elements of aa and cc at position $bb(b0 + X0)$ are equal; recall that $b0$ is the initial value of b. Note that the generated invariants have skolem functions introduced: `sk1(X0)` denotes a skolem function of $X0$.

3.2.4 Post Processing

Since we rely on a saturation engine to generate consequences from our symbol elimination problem we can notice that some of the loop invariants that are generated are redundant, that is they are implied by other invariants in the same set. For this purpose we are interested in minimizing the set of invariants by eliminating those that are redundant. In general we can define the minimization process as follows. Given a set of formulas, we try to eliminate those formulas that are redundant in this set. In other words, assuming we have a set of formulas $\mathcal{S}$ we want to find the minimal subset of formulas $\mathcal{S}'$ such that the formulas in $\mathcal{S} \setminus \mathcal{S}'$ are consequences of $\mathcal{S}'$.

In the post processing part of Lingva, we try to minimize the set of invariants by eliminating redundant ones. For doing this we again rely on Vampire's infrastructure. To this point we have generated a set of invariants from the problem we collected. Also we are assured that all these invariants are logical consequences of the initial problem and use only symbols that appear in the original loop. Since we are interested in a small set of powerful invariants that can be used in proving the intended properties we want to minimize the set of invariants. The problem that arises here is that of proving first-order properties redundant with regard to a set of properties is undecidable. Hence we decided to do minimization (redundancy elimination) based on different strategies and with time limit.

```
...
loop invariant
\forall integer X0; aa[sK1(X0)]==cc[sK1(X0)] ||
    !(X0<(b-b0)) || !(0<=X0);
loop invariant
\forall integer X2, integer X1; !(sK1(X0)=X1) ||
    !((b0+X0)=X2) || !(X0<(b-b0)) ||
    !(0<=X0) || bb[X2]==X1;

...

  while (a < m) {
       if (aa[a] == cc[a]) {
           bb[b] = a;
           b = b + 1;
           }
       a = a + 1;
       }
```

Figure 3.9: Loop k form Figure 3.3 annotated with invariants in ACSL [12] format

For our purpose we implemented a number of four different strategies each running with a default 20 seconds time limit. These strategies are basically combinations of theory-specific simplification rules and special literal selection functions controlled by options. In a sense one can see each of the strategies as a mix of different Vampire options that allow fine tuning for the purpose of consequence elimination.

After invariant minimization, by default Lingva converts the minimized set of invariants in the ACSL annotation language of the Frama-C framework [24]. But one can also specify that the output syntax should be TPTP by using the `-format tptp` option when running Lingva. The input program of Lingva is then annotated with these invariants and it either writes a file that contains the annotated program or it outputs the results in the terminal. Besides this one can also specify that the entire generated problem has to be collected by using the `-completeOutput` option. In combination to this option a folder can be specified so that for each of the analyzed loops a tptp file containing the generated properties will written. This option can be used only if the `tptp` format is used for outputting the the invariants.

Using the ACSL syntax for Lingva to output invariants makes it easier to integrate the invariants generated by us in the Frama-C verification framework. Inside Frama-C framework one can add different loops annotated with invariants and using them can

prove correctness of invariants based on the loop.

Example 3.2.5. In Figure 3.9 we present the kth loop from Figure 3.3 annotated with a subset of invariants in ACSL format. To be more precise the invariants that are annotated in this figure are the same as the ones in Figure 3.8. These invariants are actually the ones that are used in order to prove the intended property for this loop.

Proving Program Properties

In addition to the default workflow given in Figure 3.2, Lingva can be used not only for generating but also for proving properties. That is, given a program loop with user-annotated properties, such as post-conditions, one can use Lingva to prove these properties in two steps. In the first step the program analysis method is applied as presented before and invariants are generated. While in the second step Lingva tries to prove that the annotated property is a logical consequence of the set of generated invariants.

To be more precise, during the first step of the proving process, the input program, without the annotations is passed to Lingva. By doing so loop invariants are generated based on the set options. Note that for using this feature one has to specify as output language to be TPTP. Besides the output format one can ask Lingva to generate separate files for each of the loops that are analyzed. Each of these files will contain the set of invariants in TPTP format prepared for the proving step. As a preparation step Lingva takes care of changing the formulas so that each of the invariants will be used as hypothesis in the new problem.

As per the second step, the user has to manually convert the intended property (annotations in the original code) into TPTP format. After this process is done, one adds the property to be proven as a conjecture to the generated TPTP file. By doing so a new TPTP problem is created. For proving this conjecture Lingva relies again on Vampire. What happens is that Vampire gets called on the TPTP problem and tries to solve the problem. If a refutation is found it actually means that the property to be proven (user annotated) is a consequence of the generated set of invariants. Vampire also generates a proof for refutation. From this proof one can easily extract information about which invariants where used in order to prove the intended property. Example of extracted invariants that are used in proving the intended properties are presented in more details in Subsection 3.3.

Example 3.2.6. Consider the simplified program given in Figure 3.10. Note that the loop between lines 2-8 corresponds to the kth loop of Figure 3.3. The code between lines 9-11 specifies a user-given safety assertion, corresponding to the first-order property $\forall j : 0 \leq j < b \implies aa[bb[j]] = cc[bb[j]]$. This safety assertion is the property that has to hold after execution of the loop. Using the invariants generated by Lingva on this loop one can prove the intended property. For doing this one has to apply the method presented before and convert the assertion into first-order logic and then call Vampire on

```
1  int a=b=0;
2  while(a<m){
3    if (aa[a]==cc[a]){
4      bb[b]=a;
5      b = b+1;
6    }
7    a = a+1;
8  }
9  for (int j=0; j <= b-1; j++){
10   assert(aa[bb[j]]=cc[bb[j]]);
11 }
```

Figure 3.10: Program with assertion

the new problem. By doing these steps one can prove the intended property in basically no time. To prove this property Vampire makes use of two of the generated invariants. More details about the used invariants can be found in Table 3.2.

3.3 Experiments using Lingva

For evaluating Lingva we focused on two major benchmarks. First we collected a set of example programs that appear in academic research papers dealing with invariant generation, see [36,55,99]. On the other hand we also wanted to quantify the usefulness of Lingva on production source code. For the latter case we decided to collect a set of loops from different archiving packages and try to analyze them. All our experiments were conducted using a Lenovo W520 laptop with 8GB of RAM and Intel Core i7 processor running Ubuntu 64bit. The problems that we used in our experimental section are available for download from Lingva's homepage [1].

Let us start by giving an overview of the results obtained using Lingva. Table 3.1 summarizes the entire set of experiments. The first column lists the number of examples from each benchmark suite. Second column presents the number of loops collected in each of the benchmarks, while third column gives the number of problems that could be analyzed by Lingva. We say that a loop is analyzable by Lingva if it can generate invariants for that loop without user intervention. We also consider analyzable the loops that require minimal user intervention. The fourth column shows the average number

[1] Lingva homepage `www.complang.tuwien.ac.at/ioan/lingva.html`

of generated invariants, whereas the fifth column lists the average number of invariants after minimization.

Program	♯ Loops	♯ Analyzed Loops	Avg. ♯ Inv.	Avg. ♯ Min. Inv.
Academic Benchmarks [36,55,99]	41	41	213	80
Open Source Archiving Benchmarks	1151	150	198	62

Table 3.1: Overview of experimental results obtained by Lingva.

From this table one can notice that invariant generation using Lingva produces on average 200 invariants. These invariants are generated using a time limit of 5 seconds per problem. In the case where loops contain conditionals or nested conditionals the number of generated invariants increases a lot. The reason behind this increase comes from the way we formalize into first-order the input program. One can also notice that the formalization of such a loop creates many more properties. This translates into having a bigger input problem for the saturation engine. Due to such a behavior and the fact that in most practical applications when we speak about array manipulation loops contain conditional structures it is essential for us to try minimizing the set of invariants.

In our case we apply a couple of strategies, to be more precise we use the same strategies that are presented in the post processing part of Lingva. From different experiments using minimization strategies we decided that a time limit of 20 seconds or more for each strategy proves to give best results. Throughout our experiments we set the time limit for each of the minimization strategies to be 20 seconds. From our experiments we can observe that the average number of invariants that are outputted after minimization is performed decreases significantly. Table 3.1 summarizes the results obtained in our experiments. Following the results summarized in this table one can notice that indeed the number of invariants left after minimization decreases significantly. When considering the percentage of discarded invariants, one can notice that in the case of academic benchmarks 63% of the invariants are proved to be redundant. In the following we analyze in more details the results obtained using the two benchmarks.

3.3.1 Academic Benchmarks

A summary of the experiments that we have conducted can be found in Table 3.2 and 3.3. Loops that are presented here come from various research papers, for more details see [36, 55, 99]. To be more precise these research papers either handle with the problem of program verification in more general sense or explicitly invariant generation. Table 3.2 presents loops that handle arrays and contain some conditionals, while Table 3.3 presents different loops that do not contain conditionals. Since these examples come from academic research papers, each of them were already annotated with properties to be proven.

Both of the tables are organized in the following manner. On the first column we present the original loop that is to be analyzed and its origin. The program annotation that has to be proven is deprecated in second column of these tables. Finally, third column lists a subset of invariants generated by Lingva. These invariants are actually the ones used in order to prove the properties from column two. In order to collect these invariants, we investigated the refutation proofs produced by Vampire for proving the intended property, presented in column two. From this refutation proofs we collected all invariants that are used and added them to the table.

Tables 3.2-3.3 show that Lingva succeeded to generate complex quantified invariants over integers and arrays, some of these invariants using alternation of quantifiers. Although existential quantifiers do not explicitly appear in the invariants, skolem functions do.

Take the `Partition_Init` for example. Invariants generated by symbol elimination make use of skolem functions, in our example denoted by $sk1$. Now if one would apply de-skolemisation to these functions we would obtain invariants that have quantifier alternations. Recall that in order to eliminate existential quantifier one needs to introduce a new unary function that depends only on the universally quantified variables. Now the reverse process eliminates these functions and introduces existential quantified variables. But it is not always a good idea to try de-skolemizing invariants. More precise, after the de-skolemization process is done, it could be the case that the invariants are not strong enough to prove the intended properties.

Analyzed loop	Program annotation	Generated invariants implying annotation

`Partition` [99]
$a = 0; b = 0; c = 0;$
`while( a < m ){`
`if( aa[a] >= 0){`
`bb[b] = aa[a];`
`b = b+1;}`
`else{`
`cc[c] = aa[a];`
`c=c+1;}`
`a = a+1;}`
	$\forall x: \quad 0 \leq x < b \implies$
	$bb[x] \geq 0 \wedge$
	$\exists y: 0 \leq y < a \wedge$
	$bb[x] = aa[y]$
		`inv1:`
		$\forall x_0: \quad aa(sk4(x_0)) \geq 0 \vee$
		$\neg(0 \leq x_0) \vee b \leq x_0$
		`inv42:`
		$\forall x_0: \quad 0 \leq sk4(x_0) \wedge$
		$sk4(x0) < a$
		`inv81:`
		$\forall x_0: \quad \neg(0 \leq x_0) \vee b \leq x_0 \vee$
		$aa(sk4(x_0)) = bb(x_0)$

`Partition_Init` [55]
$a = b = 0;$
`while( a < m ){`
`if(aa[a]==cc[a]){`
`bb[b]=a; b=b+1;}`
`a = a+1;}`
	$\forall x: \quad 0 \leq x < b \implies$
	$aa[bb[x]] = cc[bb[x]]$
		`inv3:`
		$\forall x_0: \quad \neg(0 \leq x_0) \vee \neg(x_0 < b) \vee$
		$aa(sk1(x_0)) = cc(sk1(x_0))$
		`inv10:`
		$\forall x_0, x_1, x_2: \quad \neg(sk1(x_0) = x_1) \vee$
		$\neg(x_0 = x_2) \vee$
		$\neg(x_0 < b) \vee$
		$\neg(0 \leq x_0) \vee$
		$bb(x_2) = x_1$

Table 3.2: Experimental results of Lingva on some academic benchmarks with conditionals.

We are not aware of any other tool that is able to generate invariants with quantifier

alternations. We further note that all user-provided annotations were proved by Lingva, in essentially no time. While proving the intended properties only a subset of the generated invariants where used. Basically those are the invariants presented in column three of the tables.

Another interesting aspect of the invariants generated by Lingva is that usually the interesting invariants are generated at the beginning of the process. Take for example `Partition_Init`, in this case the interesting invariants are those used to prove the intended property. One can notice from Table 3.2 that invariants `inv1, inv42` and `inv81` are used for proving the property. Where `inv1` means that this is the first generated invariant and so on.

Analyzed loop	Program annotation	Generated invariants implying annotation

`Initialization` [55]
`a = 0;`
`while(a<m){`
`aa[a]=0;`
`a=a+1;}`
	$\forall x: \quad 0 \le x < a \implies aa[x] = 0$	`inv90:`
		$\forall x_0: \quad \neg(0 \le x_0) \vee a \le x_0 \vee aa[x_0] = 0$

`Copy` [99]
`a = 0;`
`while( a < m ){`
`bb[a]=aa[a];`
`a=a+1;}`
	$\forall x: 0 \le x < a \implies bb[x] = aa[x]$	`inv104:`
		$\forall x_0, x_1: \quad \neg(0 \le x_0) \vee a \le x_0 \vee \neg(bb[x_0] = x_1) \vee aa[x_0] = x_1$

`Init_non_const` [36]
`i = 0;`
`while(i<size){`
`aa[i]=2*i+c;`
`i=i+1;}`
	$\forall x: \quad 0 \le x < i \implies aa[x] = 2 * x + c$	`inv128:`
		$\forall x_3, x_4: \quad i \le x_3 \vee \neg(0 \le x_3) \quad c + (2 * x_3) = aa[x_3]$

`Copy_odd` [36]
`i = 0; j = 1;`
`while( i < size ){`
`aa[j]=bb[i];`
`j++;i+=2;}`
	$\forall x: \quad 0 \le x < j \implies aa(x) = bb(2 * x + 1)$	`inv206:`
		$\forall x_3, x_4: \quad j \le x_3 \vee \neg(0 \le x_3) \vee aa[x_4] = bb[2x_3 + 1]$

`Reverse` [36]
`i = 0;`
`while(i<size){`
`j=size-i-1;`
`aa[i]=bb[j];`
`i++;}`
	$\forall x: 0 \le x < i \wedge \implies aa[x] = bb[size - x - 1]$	`inv111:`
		$\forall x_4: \quad i \le x_4 \vee \neg(0 \le x_4) \vee bb[size - x_4 - 1] = aa[x_4]$

`Strlen` [36]
`i = 0;`
`while(str[i]≠0){`
`i=i+1;}`
	$\forall x: \quad 0 \le x < i \implies str(x) \neq 0.$	`inv5`
		$\forall x_0: \quad i \le x_0 \vee \neg(0 \le x_0) \vee str(x_0) \neq 0$

Table 3.3: Experimental results of Lingva on some academic benchmarks without conditionals.

Whereas in the case of invariants generated for loops that do not contain any conditionals we noticed that in most of the cases the intended property appears actually as an invariant in the generated ones. Of course the main difference to the intended property is the fact that our generated invariants are in CNF form.

3.3.2 Open Source Benchmarks

In order to evaluate Lingva also on open source software we investigated the source code of some famous archiving software, GZip [44], BZip [95], and Tar [2]. After manual inspection of the source code we have collected a set of 1151 loops that contain array manipulation. From this set of loops only 150 could be analyzed by Lingva, summary is given in Table 3.1. The number of analyzed loops seems small compared to the number of loops extracted. When further investigating the 1001 loops that could not be analyzed by Lingva we noticed why this is the case. Most of these loops are nested, in this case we cannot analyze them, we can analyze only nested conditionals. Analysis of nested loops is left for future work and investigation. Other loops implemented abrupt termination, bit-wise operations, pointer arithmetic or had some procedure calls. In the current version of Lingva we do not handle these kind of constructs. But we believe that extending and combining Lingva with more sophisticated program analysis methods, such as the ones presented in [51, 52, 99], would enable us to handle more general programs then we currently do.

On the other 150 loops Lingva successfully generated interesting invariants. These loops are mostly variations of the loops presented in Table 3.3 and Table 3.2. To this extent we noticed that in general when dealing with arrays programs mostly contain loops that do copy, initialization, shift and has been successfully evaluated implemented array copy, initialization, shift and partition operations, similarly to the ones reported in our experiments with academic benchmarks. For these examples, Lingva generated quantified invariants, some with alternations of quantifiers, over integers and arrays.

We were also interested to see how Lingva behaves on examples coming from open source software when it comes to proving program properties. To this end, we manually annotated the 150 loops with properties expressing the intended properties. In all of these cases Lingva was used and managed to prove the intended properties from the set of generated invariants. All these properties were proved in essentially no time, underlining the strength of Lingva for generating complex invariants in a fully automated manner.

3.3.3 Initial Version and Limitations

The first implementation of symbol elimination was already described in [55], by using the first-order theorem prover Vampire [72]. This implementation had however various limitations: it required user-guidance for program parsing, implemented tedious translation of programs into a collection of first-order properties, had limited support for the first-order theory of arrays and the generated set of invariants could not yet be used in the context of software verification.

Although most of these limitations are addressed in the new implementation of Lingva, there are still a couple o problems that should be addressed. The current im-

plementation is not suitable for use with nested loops. Although we are able to handle nested conditionals the problem of handling nested loops is much more difficult. Among the limitations of our tool handling nested loops is one of the hardest problems.

Besides the problem of reasoning about nested loops, automated theorem provers in general are not that well suited for arithmetical reasoning and with combination of theories. For this purpose we decided to integrate bound propagation for arithmetical reasoning in Vampire. But besides this decision procedure one can think about integration of more theories that prove to behave well in the context of SMT also inside Vampire. Another direction that could be taken is to actually create a SMT like engine inside of Vampire and make communication between the first-order part of Vampire and the SMT facile.

Some other directions that can improve the usability of Lingva are related to the user interface. From this point of view, reading the properties to be proven directly from the source file and internally converting them into TPTP formulas would greatly improve usability of Lingva among program developers. Also experimenting with Lingva and combinations of other theories than the ones we have implemented for now represents and interesting challenge.

In Chapter 4 we present the first implementation of bound propagation method. This method is meant to address the problem of having decision procedures for different theories in the context of first-order theorem provers. More precisely bound propagation is meant to improve reasoning with theories in the context of Vampire. For Lingva using a combination of different theories should greatly improve invariant quality. This is due to the fact that in general in program verification arithmetic is essential.

CHAPTER 4

Bound Propagation for Arithmetic Reasoning

The problem of solving systems of linear inequalities is a well studied problem. In this chapter we present the bound propagation method for solving systems of linear inequations. We will then continue by presenting the first ever implementation of this method in the framework of Vampire. In order to make this implementation efficient there are a couple of important points where some essential decisions have to be taken. We present in details all these points and conclude this chapter by presenting the experiments that we conducted with bound propagation.

4.1 Bound Propagation Method

Solving a system of linear inequalities over rational and/or real numbers is a well studied problem. The main methods used to solve such systems are Simplex [20] and interior point [93]. Several new methods have recently been developed in the automated reasoning and SMT community. These include the conflict resolution method [64] and GDPLL [80], and the recently introduced bound propagation method [67]. The state-of-the-art SMT solvers, such as Z3 [33] and Yices [40], use Simplex. They also use sophisticated preprocessing algorithms to simplify the input problem.

In this chapter we start first by describing the general bound propagation method introduced in [67], also denoted by BPA in the sequel. After we make a short overview of this method we continue and present details about how we managed to implement BPA in the first-order theorem prover Vampire. The chapter is concluded by presenting experiments done using the newly implemented method. When compared to Z3 and Yices, our experiments show encouraging results. In particular, BPA can solve problems

which are difficult for both Z3 and Yices, see Section 4.4. This is especially promising, given that our implementation of BPA in Vampire does not use any preprocessing steps.

In a nutshell, BPA works as follows. Given a system of linear inequalities over reals, BPA tries to iteratively assign values to some variables and, using these values, derive bounds on other variables of the problem by *bound propagation*. By a bound on a variable x we mean a linear inequality $x \geq c$ or $x \leq c$, where c is a real constant. This process either derives an inconsistent pair of bounds on some variable, or generates an assignment that solves the system. If such a pair is found, BPA builds a *collapsing inequality*, which is used to derive a new bound on a previously assigned variable v, so that the new bound excludes the value previously assigned to this variable. In some cases this new bound is inconsistent with a previously known bound on v, which means that the system is unsatisfiable. Otherwise, we backjump to the point where we selected a value for v and select a new value for v, this time satisfying the newly derived bound.

From this brief description of BPA, one may identify many similarities between BPA and propositional DPLL [82], in particular when it comes to inequality/clause learning, variable selection/ordering and backjumping. Generalizing the ideas of DPLL to arithmetic reasoning is also the key ingredient of the conflict resolution [64, 65] and GDPLL [80] method.

4.1.1 General Presentation

Let us first start by fixing the notions that are used throughout this chapter. The material of this section is based on that of [67] and adapted to our setting.

We denote variables by v, x, y, z and real constants by c, maybe with indices. We call a *literal* a variable x or its negation $-x$ and denote literals by l. By a *linear inequality* we mean an expression $c_1 l_1 + \cdots + c_n l_n + c \geq 0$, where the variables in literals are pairwise different. Throughout this chapter we consider only non-strict inequalities so that it is easier to present. Note that although here we use only non-strict inequalities, both the BPA method and our implementation, applies to systems that may contain both strict and non-strict inequalities.

We say that an inequality is *trivial* if it contains no variables. For simplicity, we assume that trivial inequalities are either $-1 \geq 0$ or $0 \geq 0$.

An assignment σ over a set of variables $\{x_1, \ldots, x_n\}$ is a mapping from $\{x_1, \ldots, x_n\}$ to the set of real numbers $\mathbb{R}$, that is $\sigma : \{x_1, \ldots, x_n\} \rightarrow \mathbb{R}$. For a linear term q over $\{x_1, \ldots, x_n\}$, we denote by $q\sigma$ the value of q after replacing all variables $x_i \in \{x_1, \ldots, x_n\}$ by the corresponding values $\sigma(x_i)$. An assignment σ is called a solution of a linear inequality $q \geq 0$ if $q\sigma \geq 0$ is true; in this case we also say that σ *satisfies* $q \geq 0$ (otherwise, it *violates* $q \geq 0$). An assignment σ is a *solution of a system* of linear inequalities if it is a solution of every inequality in the system. A system of linear inequalities is said to be *satisfiable* if it has a solution.

A *bound* on a variable x is an inequality of the form $x \geq c$, called *lower bound*, or $-x \geq c$, called *upper bound*. For example, $x \geq 0$ is a (lower) bound on x. The bounds $x \geq c_1$ and $-x \geq c_2$ are said to be *contradictory* or *inconsistent* if $c_1 + c_2 > 0$. A pair of inconsistent bounds on x is called a *conflict*. A (lower) bound $x \geq c_1$ *improves* a (lower) bound $x \geq c_2$ if $c_1 \geq c_2$. Similarly, an (upper) bound $-x \geq c_1$ *improves* an (upper) bound $-x \geq c_2$ if $c_1 \geq c_2$.

4.1.2 Resolution with Inequalities

We define resolution in the context of bound propagation similar to the way of solving upon a variable in the general settings of algebra.

Definition 4.1.1. Assume that we have two linear inequalities I_1, having the following form $d_1x + I_1' \geq 0$, and I_2, of the form $-d_2x + I_2' \geq 0$. As in the general framework of mathematics we define the resolution of these two inequalities to resolve them upon the variable x. By applying resolution on I_1, I_2 and solving upon x we obtain the following inequality $d_2I_1' + d_1I_2' \geq 0$, called *resolvent of* I_1 *and* I_2. In general we call *Resolution* the inference that takes two inequalities and derives an resolvent for them.

Example 4.1.1. Take for example the following two clauses (inequalities) : $x+3y+2z+9 \geq 0$ and $-2x-5y+4 \geq 0$. We can apply resolution on these two clauses. Depending upon which variable we resolve we get two resolvents. That is, resolving upon x we obtain $y + 4z + 22 \geq 0$, whereas if we resolve upon y we obtain $-x + 10z + 57 \geq 0$ as resolvent for the two equations. Note that any of the two resolvents are consequences of the initial inequalities. Another important feature is the fact that resolution is compatible with equivalence. That is, assuming we replace one of the premises (inequalities) by another equivalent inequality the result obtained after applying resolution will be the same as if no replacement would happen.

Assume that we now have a bound, let's call it *b*, and an inequality *I*. Applying resolution on the *b* and *I* would eliminate a variable from the inequality *I*. To be more precise it will remove exactly the variable contained in the bound. Therefore assuming we have a linear inequality *I* containing *n* variables $x_1, \ldots, x_n$ and we have a set of bounds for $x_2, \ldots, x_n$ we can apply n times the resolution step and would obtain a bound on the variable x_1. Using this idea of repeated applications of resolution we can define the bound resulting resolution as follows.

Bound Resulting Resolution

Definition 4.1.2. Assume that we have any linear inequality *I* of the form $d_1l_1 + ... + d_nl_n + c \geq 0$. And let b_i be bounds of the form $\bar{l}_i + c_i \geq 0$, where $i = 2, \ldots, n$,

on literals complementary to the literals in *I*. Then we can derive, by a sequencing resolution inferences, from $b_2, \ldots, b_n$ and I the following bound on b on l:

$$l_1 + (c + d_2c_2 + \ldots + d_nc_n)/d_1 \geq 0 \tag{4.1}$$

We will say that this bound b is obtained by *bound resulting resolution* from $b_2, \ldots, b_n$ and I.

Similar, let b_i be bounds of the form $\bar{l}_i + c_i \geq 0$, where $i = 1, \ldots, n$, on literals complementary to the literals in *I*. Then by applying the a sequence of inferences on the bounds $b_1, \ldots, b_n$ and I and obtain the following trivial inequality:

$$c + d_1c_1 + d_2c_2 + \ldots + d_nc_n \geq 0 \tag{4.2}$$

In this case we will say that the trivial inequality is obtained by the *bound resulting resolution* from $b_1, \ldots, b_n$ and I

If we now consider resolution and bound-resulting resolution as inference rules, we can put together sequences of these steps and we will obtain a *derivation*. That is a tree that contains only inferences. Take for example the derivation (1).

$$\dfrac{x + y - z - u \geq 0 \qquad u - 1 \geq 0 \qquad -y \geq 0 \qquad \dfrac{u - 1 \geq 0 \qquad z - u + 1 \geq 0}{z \geq 0}}{x - 1 \geq 0} \tag{1}$$

In this case the bound $x - 1 \geq 0$ is a derivation of a bound from two bounds $u - 1 \geq 0$ and $-y \geq 0$ and two inequalities $z - u + 1 \geq 0$ and $x + y - z - u \geq 0$. Applying multiple times bound-resulting resolution inferences we repeatedly derive new bounds for the variables. Repeated application of such rules is also known as *bound propagation*, and is formalized next.

Bound Propagation

Definition 4.1.3. Let *B* be the set of all non-trivial bounds that contain no contradiction and a *L* a system of linear inequalities. A *bound propagation* from *B* and *L* is a sequence of bounds $b_1, \ldots, b_n$, such that

1. $n > 0$
2. For all *k* such that $1 \leq k \leq n$, the bound b_k is not implied by $B \cup \{b_1, \ldots, b_{k-1}\}$
3. For all *k* such that $1 \leq k \leq n$, the bound b_k is obtained by bound-resulting resolution from $B \cup \{b_1, \ldots, b_{k-1}\}$ and an inequality in *L*.

The same definition applies in the case when the newly derived bound is a trivial inequality.

At this point one can collect all bound propagation steps in a tree and by doing so one can see the process of bound propagation as being a derivation of a bound b from B and L

Since the bound propagation method is similar to the DPLL one can see the process of bound propagation to be similar to the process of unit propagation in the DPLL procedure. The major difference between the two is that unit propagation always terminates, while bound propagation can be an infinite process.

Example 4.1.2. Assume we have the following equations:

$$\begin{aligned} x - y \geq 0 \\ y - x - 1 \geq 0 \end{aligned} \tag{4.3}$$

and the current set of bound to be $y \geq 0$. Let us demonstrate that bound propagation can be non terminating by using resolution steps. Starting from the bound $y \geq 0$ and equation $x - y \geq 0$ by applying bound resulting resolution we obtain $x \geq 0$. Now using the newly derived bound $x \geq 0$ and the equation $y - x - 1 \geq 0$ and applying resolution we obtain $y - 1 \geq 0$, which improves the original bound on y. Applying now resolution with $y - 1 \geq 0$ and the first equation $x - y \geq 0$ we obtain an improved bound $x - 1 \geq 0$. Applying again the previous steps with the current bounds we would obtain $x - 2 \geq 0$ and $y - 2 \geq 0$. By repeatedly applying these steps we can always improve the bounds on x and y, making bound propagation non-terminating.

Collapsing Inequalities

We will explain the notion of collapsing inequalities by using the example derivation from example derivation (1). In order to derive the new bound $x - 1 \geq 0$ two inferences are applied. And the bounds $u - 1 \geq 0$ and $-y \geq 0$ from the set of current bounds are used. But during the application of these two inferences another bound is derived, that is $z \geq 0$. Now let's assume that we would first resolve $x + y - z - u \geq 0$ and $z - u + 1 \geq 0$ upon z, then we will obtain $x + z - 2u + 1 \geq 0$ as result. It is obvious that this newly obtained equation does not improve any bounds. If we now use this new equation and the current bounds $u - 1 \geq 0$ and $-y \geq 0$ we can obtain the improved bound $x - 1 \geq 0$ by a single inference, as presented in 4.4.

$$\frac{u - 1 \geq 0 \qquad -y \geq 0 \qquad x + z - 2u + 1 \geq 0}{x - 1 \geq 0} \tag{4.4}$$

Instead of applying the original inferences (two) and first resolving the two equations we obtain a new equation that has the interesting property that it makes the derivation of $x - 1 \geq 0$ *collapse* into a single inference. The following theorem summarizes what we have informally explained.

Theorem 4.1.1. *Let L_1 and L_2 be two systems of linear inequalities such that $L_1 \cup L_2$ implies a linear inequality I. Then there exist two linear inequalities I_1 and I_2 such that:*

1. *L_1 implies I_1 and L_2 implies I_2;*

2. *the system $\{I_1, I_2\}$ implies I.*

Proof of this theorem and also other important properties for collapsing inequalities are found in [67, 68].

4.1.3 An Example of Bound Propagation

In this section we illustrate on a small example the main steps of bound propagation algorithm. Although a more formal presentation of the bound propagation algorithm is presented in Subsection 4.2.3, here our intention is to show how it behaves on a small system of linear inequalities in order to give the reader the flavor of BPA.

Consider the following system of inequalities:

$$\mathcal{S} = \{x - 3y \geq 2, x + 3y \geq 1, -x + y \geq 0\}.$$

Notice that in the initial step none of the variables has any bounds, hence the context (set of current bounds) $\mathcal{B}$ is initially empty. In the following we present in some details each of the essential steps of BPA.

(1) Initialize and Decide. In the first step of BPA we pick a variable x and assign to it an arbitrary value within the current bounds for this variable. The variable that is chosen to be assigned is referred as *decision variable*, while the value that is assigned to this variable is called *decision value*. Let's choose for our example the variable x. Now that we have decided which variable to assign and since in this example there are initially no bounds we choose to assign 0 to x.

(2) Bound Propagation. During this step we have do propagate all the values that have been derived in the previous step. Using $x = 0$ and the inequality $-x + y \geq 0$, we derive a new bound $y \geq 0$. The process of deriving a new bound over y using a bound on x is called *bound propagation*. Further, $y \geq 0$ and $x - 3y \geq 2$ yield a new bound $x \geq 2$, which contradicts the decision value assignment $x = 0$. Note that the contradiction was obtained using the asserted assignment $x = 0$, so we cannot yet conclude that $\mathcal{S}$ is unsatisfiable. In this case we continue by trying to analyze the conflict in recover so that the process can continue.

(3) Conflict Analysis. Using $-x + y \geq 0$ and $x - 3y \geq 2$, that is, those inequalities from S that were used to derive the contradiction, we infer an *collapsing inequality*. Remember that bound propagation algorithm works in a similar way to the DPLL procedure. That is, in case a conflict is found and unsatisfiability cannot be decided, one has to analyze the conflict and learn from it. In our case the collapsing inequality is the bound $-x \geq 1$ for x. It is be obtained by eliminating the variable y in the above inequalities.

Notice though that in general the collapsing inequality is not necessarily a bound. In our case, using the collapsing inequality yields to the exclusion of decision value 0 for variable x. Since we have to recover from the conflict, next we backjump to the decision value assignment $x = 0$. At this point we have to remove the assignment $x = 0$ and assert the new bound $-x \geq 1$ for variable x. Let us note that a collapsing inequality is always implied by S. The role of collapsing assignment in this step is that of excluding a previously decided assignment.

(4) Bound Propagation. After we have recovered from a conflict we perform bound propagation using the new bound $-x \geq 1$ and by doing so we manage to derive two new bounds for y. That is, using $-x \geq 1$ and the given inequation $x - 3y \geq 2$ we obtain $-3y \geq 3$ which yield $-y \geq 1$. And from $-x \geq 1$ and $x + 3y \geq 1$ we obtain $y \geq \frac{2}{3}$ If we now inspect the two newly derived bounds we notice that they are inconsistent, meaning we found a contradiction. Since at this stage all the derived bounds do not depend on any assignment, remember that we backjumped in the previous step, BPA terminates by reporting unsatisfiability of S.

4.2 Integrating Bound Propagation in Vampire

This section describes algorithmically the bound propagation method and how we managed to implement it in Vampire. Throughout this section, when we introduce new notions that can influence the performance of bound propagation we also present the implemented option in Vampire.

We start by giving the general overview of how BPA is integrated in the context of Vampire. Figure 4.1 presents a summary of how BPA was implemented. In the following subsections we describe in details each of the steps deprecated in Figure 4.1 as well as implementation details. One can run Vampire's bound propagation decision procedure for solving systems of inequalities by invoking the bound propagation mode of Vampire. This can be done by specifying at command line the `-mode bpa` option. When using this mode one also has a vast palette of options implemented in Vampire that allow the user to better control the behavior of bound propagation. Next we will go through the most important options and present them as well as the ideas that lead to their implementation.

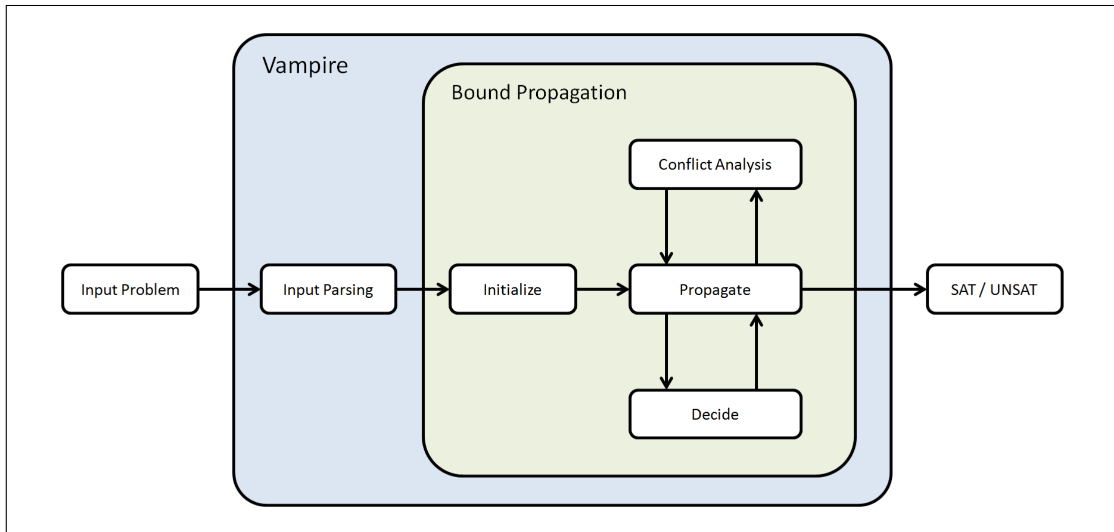

Figure 4.1: Bound Propagation for Arithmetic Reasoning in Vampire.

4.2.1 Input Problems

The input problems to BPA are systems of linear inequalities over reals or rationals. Although the bound propagation decision procedure can be used both for real and rational linear arithmetic, in our current implementation if input contains rationals they are internally converted to floating point numbers.

As input for BPA in Vampire one has to provide a conjunction of linear inequalities. This is a rather strict constraint since in general problems that deal with linear arithmetic contain also arbitrary boolean combinations. In order to address this problem we work now towards extending Vampire and BPA to accept arbitrary combinations of linear inequalities.

In general state-of-the-art SMT solvers implement a variety of preprocessing steps. These are intended both for handling the boolean structure of the problem and also for simplifying the problem. In the case of BPA and Vampire no preprocessing is done on the input problem. We focused more on assessing the power of BPA than on the preprocessing steps, hence implementing different preprocessing steps is left as future work.

Input Syntax

Problems that are accepted by Vampire with BPA must be represented in either SMT-LIB [9] format or in MIPLIB [62] format. Current state-of-the-art SMT solvers implement support for SMT-LIB format, while most of the optimization tools provide support

for MIPLIB input format. In our case we provide support for both input formats.

One can specify the format of the input problem by using the following option from command line

```
-input_syntax [value]
```

This option has three possible values: `smtlib` for SMT-LIB version 1.2, `smtlib2`, for the SMT-LIB version 2.0 and `xmps` for MIPLIB. Where the default value is set to be `smtlib`.

4.2.2 Representation of Reals

In the current implementation of BPA inside of Vampire we provide three ways of representing numbers, as follows:

(i) by using the `long double` type of C++;

(ii) by using the GNU Multiple Precision library [4];

(iii) by using our implementation of `rational`.

From a user perspective a couple options are implemented in order to allow control over the number representation. Using the option:

```
-bp_start_with_precise [on/off]
```

allows one to choose the use of precise number representation from the beginning. In a similar manner, for representing the numbers using our rational numbers implementation one has to use the option:

```
-bp_start_with_rational [on/off]
```

Note that these two options are mutually exclusive.

When a satisfying assignment is found, it is checked whether it is a solution to the original constraints using the multiple precision representation. If it is not and the `long double` or `rational` representation is used, Vampire restarts the search using the multiple precision representation.

Although it is possible to build proof objects when BPA derives the contradiction, this is not yet implemented and is left as future work. When unsatisfiability is detected, no attempt is made to check whether the result is correct, since currently Vampire does not build a proof object.

4.2.3 Arithmetic in Vampire

Given an input problem, we first translate this problem into the internal input format of Vampire. To reason about the reals, we had to extend Vampire with typed terms and formulas. We also added the built-in sort for reals and built-in functions and predicates over them, including multiplication, addition and standard comparison operators. But these additions are also used in other contexts inside of Vampire. For example in standard theorem proving mode of Vampire all these additions are used together with an axiomatization of reals.

Using these theory specific extensions of Vampire, we translate the input problem into the internal typed Vampire representation. After the translation is done we prepare the process of bound propagation by setting all the parameters according to the options and run bound propagation on it.

In the following we will go over the bound propagation algorithm implemented in Vampire.

Bound Propagation Algorithm

A pseudo-code for our bound propagation algorithm is given in Algorithm 4, highlighting the main steps of our BPA implementation. To resolve conflicts (i.e. inconsistent bounds), Algorithm 4 calls Algorithm 5.

Based on Algorithms 4 and 5, we now overview the main steps of BPA. Given a system $\mathcal{S}$ of linear inequalities, BPA searches for a solution by applying bound propagation, variable decision and conflict analysis until either a satisfying variable assignment is found, or otherwise a contradiction is derived.

Bound propagation incorporates important DPLL optimization like backjumping, lemma learning and propagation of bounds, although with essential differences. For example, our analogue of the DPLL learned clause is the collapsing inequality. Unlike in the case of DPLL collapsing inequalities are not added to the list of inequalities but rather used for backjumping. In the following we focus on the main steps of our bound propagation implementation. We also highlight main differences between DPLL and BPA in our presentation.

Initialize

We start by collecting initial bounds for variables from $\mathcal{S}$, this step is deprecated in line 3 of Alg. 4. Let us denote by $\mathcal{B}$ the set of obtained bounds. Notice the fact that we do not assume that each variable has an initial bound. Take the example from Section 4.1.3 where we initially have $\mathcal{B} = \emptyset$ and the set $\mathcal{S}$ initially contains no bounds.

Throughout the run of the BPA algorithm, the set $\mathcal{B}$ will be changing. In addition to the bounds, we also store *assignments* of values to decision variables that we make during the run of BPA. An assignment is represented by an equality $v = c$, which is also

Algorithm 4 The BPA Algorithm

```
1:  Input: a set of linear inequalities S
2:  Output: satisfying assignment/"unsatisfiable"
3:  (Initialize) collect input bounds B ;
4:  set decision level DL := 0;
5:  while no solution found do
6:      (Propagate) propagate bounds;
7:      if conflict then
8:          (Conflict Analysis)
9:          call Conflicts Analysis ( Alg. 5);
10:     else
11:         if there exist variables without decision then
12:             (Decide)
13:             DL := DL + 1;
14:             select next decision variable v (the decision
15:                 variable of level DL);
16:             select a decision value d for v;
17:             add decision bounds v ≥ d and −v ≥ −d
18:                 to B;
19:         else return the map from the variables to
20:                 their decision values as a solution;
```

Algorithm 5 Conflict Analysis in BPA

```
1:  while there are conflicting variables do
2:      build the collapsing inequality CI;
3:      if CI is −1 ≥ 0 then
4:          return "unsatisfiable";
5:      CV := variable in CI with the maximal decision level;
6:      DL:= decision level of CV;
7:      backjump to the decision level DL;
8:      add the new bound on CV generated by CI to B;
9:      select a new decision value for CV;
10:     add new decision bounds on CV to B;
```

treated as a pair of bounds $v \geq c$ and $v \leq c$ in $\mathcal{B}$. The set of assignments is assumed to be initially empty/

The decision level (DL) of BPA corresponds to the number of decision value assignments to variables. Decision level in the context of BPA is similar to the decision level in the context of DPLL. In the case of BPA as in the case of DPLL we start from decision level 0, line 4 of Alg. 4. For more information about how and when we increase or decrease the decision level see the *Decide* step.

Propagate

We use the bounds in $\mathcal{B}$ to derive new bounds (line 6 of Alg. 4), which are added to $\mathcal{B}$. The new bounds are logical consequences of $\mathcal{B}$ and $\mathcal{S}$. The process of deriving new bounds using other bounds is called *bound propagation*. Unlike DPLL, bound propagation can become non-terminating, for example by deriving ever improving bounds for the same variable (e.g. $x \geq 0$, $x \geq 1$, $x \geq 2$, etc). For this reason, in our BPA implementation we perform *limited bound propagation*, as described below.

If bound propagation derives an inconsistent pair of bounds $x \geq c$ and $x \leq d$ with $c > d$, then we proceed to *Conflict Analysis* (line 9 of Alg. 4). Otherwise, if some variables are not assigned, we move to the *Decide* step and select the next decision variable and a value for it (lines 13-17 of Alg. 4). Finally, if all variables of $\mathcal{S}$ are assigned, we report satisfiability of $\mathcal{S}$ and output the assignment (line 19 of Alg. 4).

Limited bound propagation

To ensure termination of BPA, we perform limited bound propagation in our BPA implementation (line 6 of Alg. 4). To this end, we use the option

`-bp_bound_improvement_limit` k

to specify the upper limit k on the number of improved bounds derived by bound propagation on the same variable at the current decision level. The value k is a positive integer, its default value is set to 3. Bound propagation will terminate as soon as some variable's bound was improved k times.

Our choice of the default value was motivated by our experiments: it turned out that in most cases value 3 (and sometimes 2) yields the best results.

Conflict Analysis

At this stage we derived an inconsistent pair of bounds on a variable (line 9 of Alg. 4). We analyze these inconsistencies (conflicts), as summarized in Algorithm 5. If there are no assigned variables, we report unsatisfiability (line 4 of Alg. 5). Otherwise, we analyze the derivation of these bounds and compute the so-called *collapsing inequality*

of this derivation (line 2 of Alg. 5). The collapsing inequality is implied by $\mathcal{S}$ and can be used to derive a bound b on a previously assigned variable x such that b contradicts to the existing assignment for x. We backjump by removing all assignments made since the assignment to x and all bounds derived using these assignments (lines 5-8 of Alg. 5). We then add b to $\mathcal{B}$ (lines 9-10 of Alg. 5) and go to the *Propagate* step. It is possible to modify this method by using the collapsing inequalities also in the propagation step. This means that during the propagation step one has to take into consideration also the collapsing inequalities. This idea is similar to clause learning used by SAT solvers. One can enable this method by:

```
-bp_add_collapsing_inequalities on
```

During our experiments we noticed some improvement by adding collapsing inequalities, but they can also cause slowdown because bound propagation becomes more expensive.

Decide

At this stage, we have no inconsistent bounds but satisfiability of the system is not yet established (line 11 of Alg. 4). We therefore pick an unassigned variable x. Since the set of bounds $\mathcal{B}$ is satisfiable, so is the current set of bounds for x. We assign to x a *decision value* satisfying these bounds, increment the decision level and go to the *Propagate* step again (lines 13-17 of Alg. 4).

4.3 Strategies for Variable and Value Selection

When selecting variables and their values, our BPA implementation uses no a priori fixed variable ordering. Picking the "right" variable and choosing its value during bound propagation clearly affects the efficiency of BPA. We studied and implemented several strategies for variable and value selection. Some of these strategies were inspired by SAT solving heuristics [81]. In the rest of this section we discuss the strategies implemented for both variable and value selection. Also in the context of algebraic computation the effects of variable ordering was also well studied, see [21, 37]. But we did not yet implement these heuristics in BPA, this is left as an interesting task of future work.

4.3.1 Variable Selection

In this subsection we discuss different variable selection strategies for BPA implemented in Vampire. To specify the variable selection strategy in Vampire, one should use the option:

```
-bp_variable_selector [value]
```

where `value` specifies which strategy should be used. If this option is omitted, BPA uses the default random variable selection.

In the sequel we will refer to yet unassigned variables as *eligible variables.* Below we describe various values for the variable selection option of BPA:

`random` : In case this option is used, the strategy picks a random variable from the set of eligible variables.

`first` : This strategy selects the first eligible variable. This first eligible variable is selected given some fixed predefined order on variables. In our implementation this ordering is basically the order variables appear in the problem.

`tightest_bound` : Pick an eligible variable with the tightest bound. Thigthness of a bound is computed by doing the difference between the upper and lower bound of a variable. This strategy has the advantage of selecting variables which are more likely to derive either a contradiction or to allow fast process in the search.

`conflicting` : This strategy, inspired by the VSIDS heuristics of [81], picks an eligible variable which appears most often in conflicts. That is we keep track of all variables that appear in conflicts and at the next decision step we pick the one which appeared more often.

`conflicting_and_collapsing` : Pick an eligible variable which appears most often both in conflicts and collapsing inequalities.

`recent_conflicting` : This strategy allows us to pick an eligible variable which appears most often in recent conflicts.

`recent_collapsing` : Using this strategy we select an eligible variable which appears most often in recent collapsing inequalities.

4.3.2 Assigning Values to Variables

Different *decision values* can affect both the number of steps performed by the algorithm and the size of the numbers involved in bound computations. Since both items are important for efficiency of BPA algorithm, we implemented several strategies for value selection.

To specify the variable value selection strategy in Vampire one should use the option:

```
-bp_assignment_selector value
```

where `value` specifies which strategy should be used.

In addition to the value selection, we also implemented a mechanism to keep track of previously assigned values to variable, which can be toggled using:

```
-bp_conservative_assignment_selection [on/off]
```

The conservative value selection was considered in order to speed up the value selection computation. By default, this option has the value `off`. When this option is `on`, we store the history of all variable assignments. Upon the backtrack in BPA (see Alg. 5), we first try to find an old value of the variable which satisfies new bounds and only if there is no such value we choose a new value for this variable. Of course this value is choosen according to the strategy specified the option that controlls which value selector is in use:

```
-bp_assignment_selector [value]
```

In what follows we describe some of the BPA strategies for assigning values to decision variables. The value that has to be passed to this option in order to activate each of the described options is in parantesis after its name. For simplicity, if a variable has no upper bound, we will sometimes consider ∞ as its upper bound, and likewise $-\infty$ for the lower bound. We will call the *interval* for a variable the set of all values between the lower and the upper bound.

Random value assignment (`random`) As the name suggests this strategy tries to select a random value for the variable. If the variable has both bounds, the strategy will pick a random value between these bounds. In case the upper bound is missing, the algorithm behaves as if some large positive number was the upper bound. Similar for a missing lower bound the algorithm behaves as if some really large negative number was the lower bound. This strategy is currently the default one.

Smallest absolute value assignment (`smallest`) This strategy picks the value with the smallest absolute value between the lower and upper bounds of the variable. If 0 belongs to the interval described by the bounds of the variable that we are trying to assing then we pick 0. Now in case both bounds are positive, we pick the lower bound as the variable value (this just in the case of nonstrict bounds), otherwise we pick the upper bound. If the selected lower bound is strict, we add a small number δ to it, and likewise for the upper bound.

Alternating lower and upper bound (`alternating`) This strategy implements alternations of picking a lower or upper bound of a variable. In case when the bound on the variable is strict, the variable value picked is the bound plus/minus some small value δ. If the selected bound is missing, a large number is taken instead.

There are also similar values `lower_bound` and `upper_bound` for this option. For example, when `upper_bound` is used, the behavior is as follows. Where is no upper bound, a large positive value is chosen. If there is a non-strict upper bound, this bound is chosen. Otherwise when the upper bound b is strict, we choose $b - \delta$ for some small δ.

Middle value (`middle`) This strategy picks the arithmetic mean of the interval defined by the lower and upper bound of a variable. If the upper bound is missing some very large positive value is used instead, and similar for the lower bound.

Tight (`tight`) This strategy picks a value such that it differs from the upper or lower bound by a small δ. If both bounds are present, we randomly choose which bound to use. If none is present, we choose 0 as value for the variable.

Binary decomposition (`bmp`) Having a lower and upper bound for a variable, the strategy starts by computing the arithmetic mean of the integer approximation of the bounds. That is, the arithmetic mean of the floor and ceiling of respectively the lower and upper bound is calculated. Having computed this mean, before assigning it to the variable we have to test whether it is in the interval described by the bounds. If this value lies within the interval bounding the variable then we assign it to the variable. Otherwise, if this value is greater than the upper bound, we compute the arithmetic mean of this value and the floor of the lower bound and re-iterate the process of checking whether the value belongs to the interval. If the value is smaller than the lower bound, we compute the arithmetic mean of the interval defined by this value and the ceiling of the upper bound, and re-iterate the process again. Main advantage of using this strategy in order to assign the value of a variable is that it uses division by 2. In modern computer architectures division by 2 is cheap and also precise up to a certain limit.

Continued fraction decomposition (`cfd`) This strategy is based on the continued fraction decomposition of rationals. The method implemented is based on iteratively representing the number as the sum of integer part and the reciprocal of the fractional part. In order to pick a good value in between two numbers one has to iteratively decompose both numbers and stop at the point where the integer part of the numbers first differ. When this happens we start computing the result of fractions obtained until that point. The procedure ensures that we are always picking a rational value with the smallest denominator and numerator among all rationals in the interval. Using this method we can pick the value with the best rational representation in the interval.

In particular, this method always picks an integer value, if one exists in the interval. This is convenient due to the fact that working with integer numbers is much less expensive in terms of speed and memory consumption. The major bottleneck associated to

this method is that it requires in some cases a considerable number of extra computation steps. This just in order to decompose the bounds. Our implementation of the method is designed in such a way that cheap operations are preferred, meaning that we are trying to avoid division and multiplication as much as possible.

4.3.3 Using Vampire's BPA

To use BPA in Vampire for solving systems of linear inequalities, one should execute the following command:

```
vampire -mode bpa problem
        -input_syntax [smtlib/smtlib2/xmps]
        -bp_start_with_precise [on/off]
        [Optional Parameters]
```

where the input parameter `problem` is the problem to be solved in the corresponding syntax. In order to control how BPA initially represents the numbers one has to enable the following flag:

```
-bp_start_with_precise [on/off]
```

By using this flag one ensures that BPA starts computing using precise number representation. More precise, using this flag enables BPA to represent numbers using the GMP precise number representation. In the same manner one can use

```
-bp_start_with_rational [on/off]
```

option which enables BPA to start computing using our rational number implementation. The `Optional Parameters` are different combinations of options, as presented in Section refsec:strategies. In a nutshell one can see these options as being a good way of controlling the way variables and their value are selected. As well as a good way of obtaining different strategies fine tuned for different problems.

For controlling the time limit when running Vampire with BPA, one should use the

```
-time_limit seconds
```

option. The default time limit of BPA is 60 seconds. Of course in case one has different application domains this time limit can and should be changed accordingly.

Our BPA implementation adds an arithmetic decision procedure to Vampire. We extended Vampire with the new built-in sort of reals and built-in theory symbols for them. We added theory axiomatisations for these symbols and extended Vampire with typed first-order formulas. We also changed the SMT parser of Vampire to read SMT-LIB problems for linear real arithmetic. Further, we implemented the BPA algorithm as described above. As for data structures and memory management, we used the standard libraries, data structures and memory allocations functions of Vampire [72]. All together, the BPA implementation in Vampire contains about 8500 lines of C++ code.

Combinations of Strategies

During experiments we extensively evaluated many different combinations of the previously described options. In many cases different combinations solve different sets of problems. Some combinations of options prove to solve more problems than others. Let us note that, even if a strategy solves only a small number of problems, we cannot qualify it as a bad strategy since some problems could be solved only by this strategy.

There are however a few combination of strategies which turned out to perform the best in most of our experiments. For example, the combination of the assignment selector `tight`, `cfd` and the variable selector `tightest_bound` manages to find all unsatisfiable cases, but on satisfiable cases does not give the best results. Although in general this strategy is not the best for satisfiable problems, we found a few examples that could be solved only by these value selections strategies. Using `cfd` in combination with internal conversion of native numbers to rationals or precise proves to solve most of the problems. Experimenting with strategies which dynamically adjust options during the run of the algorithm is an interesting task to be investigated as future work.

4.4 Experiments using Vampire's new Decision Procedure

In this subsection we present the experiments done using BPA's implementation in Vampire. For the entire set of experiments we decided on a 60 seconds time limit. Experiments where conducted on the Infragrid infrastructure from West University of Timisoara [5]. The infrastructure is build from 100 machines and each of them is equipped with an Intel Quad-core running at 2.00 GHz frequency and 14GB of RAM per CPU.

Benchmarks

We evaluated our implementation of bound propagation in Vampire on three sets of problems as presented below.

(i) We ran BPA on 128 problems generated by using the Hard Reality tool [66]. These problems were generated by the tool by using the QF_LRA benchmark suite of the SMT-LIB library [9]. The reason why we could not directly run BPA on SMT-LIB problems is that SMT-LIB examples have a non-trivial Boolean structure. Hard Reality Tool is basically a tool that extracts random hard and realistic theory problems, from SMT problems with non-trivial structure. The output of Hard Reality are problems that are basically conjunctions of constraints, that is the problems in the supported input for Vampire with BPA. In order to achieve

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	2797	0.362	17100	0.236	1576
Z3	3193	0.072	18205	0.144	75
Yices	3206	0.001	18267	0.001	0

Table 4.1: Experiments on problems generated with GoRRiLa.

this Hard Reality uses an SMT solver in the background and extracts from the input problem systems of linear inequalities that are hard for the background SMT solver.

(ii) We generated 21,473 hard random linear arithmetic problems using the GoRRiLa tool [66]. GoRRiLa is a tool that allows us to generate random linear arithmetic problems and propositional problems according to some user specifications. In our case we decided to generate problems containing on average 60 variables and 100 inequalities. By using these specifications for GoRRiLa we managed to generate a large set of problems some of them hard even for the state-of-the-art SMT solvers.

(iii) In order to evaluate our approach also on problems coming from different research fields we decided on using some of the benchmarks from linear optimization field. For this reason we extracted a number of 224 linear optimization problems from the MIPLIB library of mixed integer problems [62]. The problems basically model different optimization problems, most of them coming from industry. Although these examples contain both integer and boolean variables we treat them as real variables. That is for each boolean problem we add them as being reals and we add as lower bound for these variables to be 0 while the case of upper bound is set to be 1.

Vampire with bound propagation and the examples used for our experiments can be downloaded from bound propagation webpage: `http://www.complang.tuwien.ac.at/ioan/boundPropagation`.

Since BPA implementation in Vampire is a new method for solving systems of linear inequations we decided to compare against state-of-the-art solvers like Z3 and Yices. Both these solvers implement some variations of the Simplex-based methods. When we ran our experiments the same time limit of 60 seconds was set also for Z3 and Yices. In case of Z3 we used version **3.2** while for Yices we used version **1.0.36**.

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	33	5.23	18	3.78	77
Z3	76	2.89	22	3.40	30
Yices	85	9.90	26	6.02	17

Table 4.2: Experiments on the SMT-LIB problems generated with HardReality.

Strategies

For better understanding how different options influence the performance of BPA we experimented with various combinations of options. In this context we call a combination of options to be a *strategy* for BPA. For providing a good cover of options we devised different strategies that include random, tight, conflicting, collapsing and conflicting variable selectors. As well as random, middle, binary decomposition, continued fraction decomposition and alternating value selectors. Since we know that bound propagation can be infinite and that the limit imposed to bound propagation influences the performance we decided to evaluate our implementation by using a limit of 3, 4, 5 and 6 propagations.

In order to have a more broad view of how different strategies influence results on hard problems, we did a comparison among all the strategies using the MIPLIB benchmarks, Figure 4.2. From the experiments we notice that choosing a good assignment selector makes a big difference when compared with the default one. In terms of assignment selector, best results were obtained using the continued fraction decomposition (**cfd**). Alongside this selector, also the one based on binary decomposition (**bmp**) selector and the upper bound (**upper_bound**) selector proved to perform well. Variable selector plays also an important role in performance. In this case tightest bound (**tightest_bound**) variable selector proves to perform best.

We also observed that choosing an appropriate value as the upper limit of bound propagation `-bp_bound_improvement_limit` option is essential. From the one to one comparisons among options we observed that our decision procedure performs best in case the upper limit for propagation is set to be 2, 3 or 4. Although these values are not universally best performing, we noticed that in general choosing a propagation bound limit higher than 4 does not improve the number of solved problems.

Results

Tables 4.1 ,4.2, 4.3 summarize the results we obtained. All the tables are organized in the same way, that is, the first column lists the name of the solver. Columns two and four show the number of problems whose satisfiability (SAT), respectively unsatisfiability (UNSAT) were proved, whereas columns three and five give the respective average

solving times in seconds. Column six lists the number of problems that could not be solved within 60 seconds.

We started by evaluating Vampire with BPA on 21,473 random hard theory problems generated by GoRRiLa. The number of variables in these problems was around 60, and each problem contained about 100 inequalities. Table 4.1 summarizes the experiments we have conducted with the problems generated by GoRRiLa. Unlike the 128 SMT-LIB experiments, in this case we evaluated Vampire using BPA only with the default values for options. All together, Vampire solved 19,897 problems with a very small average solving time, and timed out on 1576 problems. Not surprising all the problems that could be solved by Vampire could also solved by Yices. However they were 75 problems that could be solved by Vampire but not be solved by Z3. Besides the fact that these problems could be solved by Vampire also the time required to do so was small, on average 0.01 seconds. But it turned out that also Z3 can solve all them if the time limit is increased to 240 seconds.

Further, Table 4.2 describes our results on the 128 SMT-LIB problems that we tried. In this case, the 51 problems that where solved by Vampire using BPA proved to be a subset of the problems solved by Z3 and Yices. Also, these 51 problems are not all solved by a single strategy for BPA, but rather the total number of problems that Vampire with BPA managed to solve. Throughout these experiments we have deployed approximately 3000 strategies for Vampire using BPA. This big number of strategies allowed us to fine tune BPA and also better understand how each of the options influences the overall result. There where also 77 problems that could not be solved using this decision procedure implementation. We believe that a relatively weak performance of Vampire on these benchmarks is due to the absence of preprocessing. By further investigation on the problems we noticed that the benchmarks contain many redundancies. Hence if preprocessing is implemented for this decision procedure we believe that the performance of Vampire with BPA will significantly increase.

Finally, Table 4.3 describes our results on 224 problems taken from the MIPLIB library. These examples encode optimization problems coming from academic and industrial applications of mixed integer linear programming benchmark suite. The MIPLIB problems we used contained thousands of variables and inequalities, each problem being over a few MB in size. Since some of these problems contain integer and/or Boolean variables, we created their relaxations, where all variables are treated as real. Further, we converted optimization problems into corresponding satisfiability problems.

Since the problems are in a different format than the one accepted in general by state-of-the-art SMT solvers we decided to interface Vampire with a MIPLIB parser. By doing so and adding an SMT-LIB printer to Vampire we managed to convert the problems into SMT-LIB v1.0 format that is accepted by the other solvers as well. Applying this technique we managed to convert a set of 224 problems on which we can experiment with all three solvers. As in the previous experiments, results presented in the table

Solver	Sat	Avg. Time	Unsat	Avg. Time	Unknown
Vampire	79	6.43	28	4.54	117
Z3	96	6.20	25	2.13	103
Yices	103	3.44	26	0.31	95

Table 4.3: Experiments on MIPLIB problems.

represent the total number of problems which Vampire was able to solve.

In the case of these benchmarks Vampire using BPA managed to solve in total 107 problems. The problems that where solved by Vampire contained in average 1526 variables and several thousands of inequalities. Among these problems we found 8 that could not be solved by Z3 within the time limit of 60 seconds but can be solved by Vampire. Notice that in the case of MIPLIB experiments we also deployed approximately 3000 strategies for Vampire using BPA. In order to obtain these results we ran Vampire with the same combination of strategies as for problems presented in Table 4.2. Another interesting aspect is that Vampire using BPA proved to perform better than Z3 and Yices on unsatisfiable problems.

Summarizing, Table 4.3 shows that the first implementation of BPA in Vampire gives competitive results when compared to state-of-the-art SMT solvers and performs relatively well on very large examples coming from applications.

Comparison of Strategies

We also compared the performance of Vampire with BPA using different strategies on the MIPLIB examples. Figure 4.2 summarizes our results. X-axis of Figure 4.2 represents the total number of solved instances by each of the strategies. While Y-axis represents the average time needed for Vampire with BPA to solve those instances. We also computed and plotted the Pareto line for minimization of average time and maximization of solved instances. On this graphic the strategies that appear in the lower right corner represent best performing strategies. This is due to the fact that they solve a large number of problems in the smallest ammount of time.

From Figure 4.2 one can observe which are the strategies that perform well in the case of MIPLIB problems, at least on the subset we tested. In our case using a combination of `tightest_bound` variable selector and `bmp` assignment selector and doing only 2 propagation updates per variable before bound propagation proves to be a good choice. Using this strategy, we actually managed to solve the highest number of problems. A variation of these strategies where we use `tight` instead of `bmp` and 3 propagation updates also performs relatively well when it comes to solving time of Vampire with BPA.

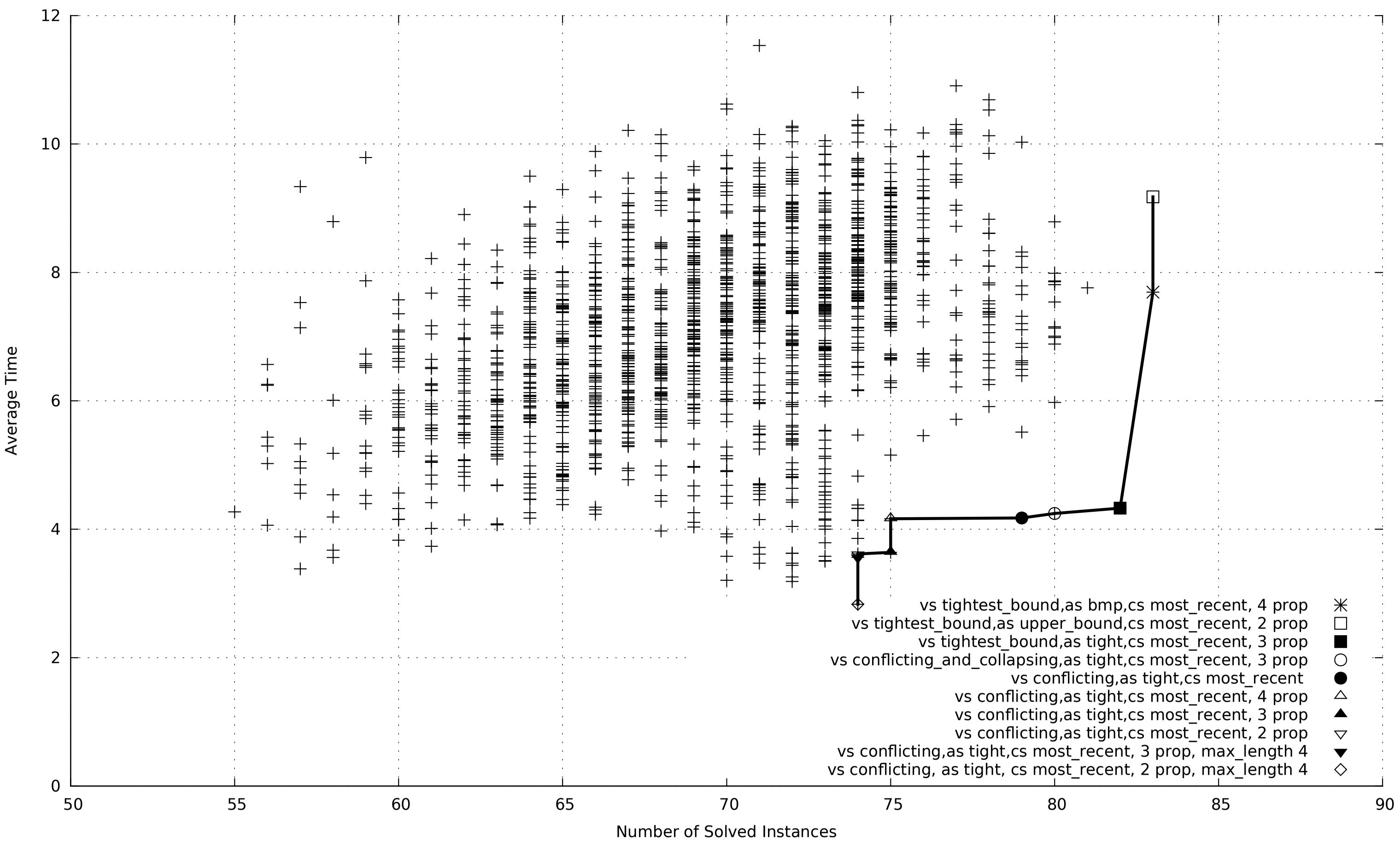

Figure 4.2: Comparison between different strategies.

Further Improvements

Our bound propagation decision procedure implementation in Vampire requires further experimentation and improvements. In particular, we are interested in (i) finding performant heuristics for choosing "good" values for options such that we better address different classes of problems. (ii) Another important issue is raised by the use of strategies for bound propagation. One can find some inspiration for this process in the literature regarding SAT solvers development. (iii) Extending the parsers of Vampire so that one can start with constratins that contain rational numbers. By doing so we are gain both time that otherwise is spent in internal transformation of numbers and also in precision. (iv) Another interesting problem is generation of collapsing inequalities. Currently we have a naive implementation for generation of collapsing inequalities. If one would invest more time in finding better collapsing inequalities, we believe that the entire process of solving systems of linear inequalities can be speed-up significantly. (v) Implement preprocessing techniques.

We believe that proper settings for (i)–(v) can speed up Vampire using BPA by orders of magnitude. The exact same thing happened in the context of SAT solvers and SAT solving techniques in the past years. Finally, one should investigate the best ways of running bound propagation in an SMT solver. While bound propagation has the main advantage that its steps can be interleaved with the SAT solver steps. Even backjumping and constraint propagation can be interleaved with SAT solving steps.

CHAPTER 5

Use of SAT Solvers in Vampire

Recently, a new reasoning framework, called AVATAR, integrating first-order theorem proving with SAT solving has been proposed. In this chapter we overview the new AVATAR architecture for first-order theorem provers. We will then continue with presenting how we managed to integrate different SAT solvers in Vampire. Interestingly, our experiments on first-order problems show that using the best SAT solvers within AVATAR does not always give best performance. There are some problems that could be solved only by using a less efficient SAT solver than Lingeling. However, the integration of Lingeling with Vampire turned out to be the best when it came to solving most of the hard problems.

5.1 AVATAR Architecture

This chapter aims to experimentally analyze and improve the performance of the first-order theorem prover Vampire [72] on dealing with problems that contain propositional variables and also other clauses that can be splitted. The recently introduced AVATAR framework [103], proposes a way of integrating a SAT solver in the framework of an automatic theorem prover. The main task that a SAT solver has in this framework is that of helping the theorem prover in splitting clauses. Although initial results obtained by using this framework in Vampire proved to be really efficient, for details see [103], it is unclear whether efficiency of AVATAR depends on the efficiency of the used SAT solver. We will address this problem using various SAT solvers and experimentally evaluate AVATAR as follows. First integrate the Lingeling [3,17] SAT solver inside Vampire and then compare its behavior against a less efficient SAT solver already implemented in Vampire.

Splitting clauses is a well studied problem in the community of automated theorem provers. The first method was introduced in the SPASS [105] theorem prover. In that case SPASS tries to do splitting and uses backtracking to recover from a bad split. Another way of dealing with splittable clauses was introduced in Vampire [89] and takes care of splitting without backtracking. Both ways of splitting on a clause are highly optimized for the theorem prover that introduced them. Implementing one or the other splitting technique is not trivial and it can highly influence the overall performance of the theorem prover. An extensive evaluation on how different splitting techniques influence performance of a theorem prover can be found in [57]. In the case of first-order theorem provers the use of splitting in general proves to help improve performance of the solver, these techniques are not as performant as the ones deployed by a SAT solver or even the ones used in an SMT solver on ground instances [103].

The problem of dealing with splitting clauses in AVATAR is motivated by the way first-order theorem provers usually work. In general first-order provers make use of three types of inferences: *generating, deleting* and *simplifying* inferences. In practice, using these inferences one can notice a couple of problems.

- Usually the complexity for implementing different algorithms for the inference rules are dependent in the size (length) of the clauses they operate on. As an example of simplifying inference, subsumption resolution is known to be NP-complete and the algorithms that implement it are exponential in the number of literals in a clause.

- Another issue arises when we want to use generating inferences. In this case assuming we have two clauses containing l_1 and l_2 literals and we apply resolution on them then the resulting clause will have l_1+l_2-2 literals. Now if these clauses are long it means we generate even longer clauses. This also raises the question of storage for these clauses for example by indexing [84].

- They are a couple of methods that deal with large clauses, for example limited resource strategy [90] which is also implemented in Vampire. This method will start throwing away clauses that slow down the prover. An alternative would be to use splitting in order to make the clauses shorter and easy to be manipulated by the prover.

5.1.1 Setup for AVATAR

This section overviews the main notions that are used in order to define the AVATAR framework, for more details we refer to [72, 103]. In the framework of first-order logic, a *first-order clause* is a disjunction of *literals* of the following form $L_1 \vee \cdots \vee L_n$, where a literal is an atomic formula or the negation of an atomic formula. Usually when we speak about splitting we speak about clauses as being sets of literals. By using this

definition for a clause we can safely assume that we do not have duplicate literals in the same clause. We also assume that predicates, functions are uninterpreted and the language might contain the equality predicate ($=$).

Splitting

In a nutshell the notion of splitting for clauses starts from the following remark. Suppose that we have a set $\mathcal{S}$ of first-order clauses and $C_1 \vee C_2$ a clause, such that the variables of C_1 and C_2 are disjoint. Then $\forall(C_1 \vee C_2)$ is equivalent to $\forall(C_1) \vee \forall(C_2)$. This transformation implies that the set $S \cup \{C_1 \vee C_2\}$ is unsatisfiable if and only if both $S \cup \{C_1\}$ and $S \cup \{C_2\}$ are unsatisfiable. In practice one can notice the fact that splittable clause usually appear when theorem provers are used for software verification purposes.

Let $C_1, \ldots, C_n$ be clauses such that $n \geq 2$ and all the C_i's have pairwise disjoint sets of variables. We can safely say that $SP \stackrel{\text{def}}{=} C_1 \vee \cdots \vee C_n$ is splittable into components $C_1, \ldots, C_n$. We will also say that the set $C_1, \ldots, C_n$ is a splitting of SP. An example of such a splittable clause can be considered any ground clause that contains multiple literals.

One problem that arises in splitting is the fact that there are multiple ways of splitting a clause. But this is not a major issue since we know that there is always a unique splitting such that each component cannot be splitted more. We call this splitting of a clause maximal. Computation of such a splitting proves to always give the maximal number of components of a clause, see [89] for details.

FO to SAT Mapping

Let us first discuss how the mapping between the first-order problem and the propositional problem is done. In the propositional problem that is sent to the SAT solver we basically keep track of clause components. In order to do that we have to use a mapping $[.]$ from components to propositional literals. The mapping has to satisfy the following properties:

1. $[C]$ is a positive literal if and only if C is either a positive ground literal or a non-ground component;

2. for a negative ground component $\neg C$ we have $[\neg C] = \neg[C]$;

3. $[C_1] = [C_2]$ if and only if C_1 and C_2 are equal up to variable renaming and symmetry of equality.

In order to implement this mapping Vampire uses a *component index*, which maps every component that satisfies the previous conditions into a propositional variable $[C]$. And for each such component C the index checks whether there is already a stored

component C‘ that are equal than it returns $[C‘]$ as propositional variable. Doing so we ensure that we do not have multiple propositional variables that are mappings of equal components.

In case there is no such component stored in the index, than a new propositional variable $[C]$ is introduced and we store the association between C and $[C]$. A model provided by the SAT solver for the propositional problem is considered a component interpretation. Such a model contains only variables of the form $[C]$ or their negations and does not contain in the same time both a variable and its negation.

The truth definition of a propositional variable in such an interpretation is standard. With the small difference that in case for a component C neither $[C]$ not $\neg[C]$ belongs to the interpretation, than $[C]$ is considered *undefined*, meaning it is neither true nor false.

AVATAR in a Nutshell

The first-order reasoning part works as usual, using a saturation algorithm [72]. The main difference with respect to a classical approach is the way it treats splittable clauses. A general overview of this process is deprecated in Figure 5.1. In the case that a clause $C_1 \vee C_2 \cdots \vee C_n$ is splittable in $C_1, C_2, \ldots, C_n$ components and the clause passes the retention test it is not added to the set of *passive* clauses. Instead we add a clause $[C_1] \vee [C_2] \vee \cdots \vee [C_n]$ to the SAT solver and check if the problem added to the solver is satisfiable. If the SAT solver returns unsatisfiable, it means that we are done and report it to the first-order reasoning part. In case the problem is satisfiable, we ask the SAT solver to produce a model. This model acts as a component interpretation *I*. If in the interpretation a literal has the form $[C]$ for some component C then we pass to the first-order reasoner the component, where C is used as an assertion. Exception from this rule are those literals of the form $\neg[C]$, where *C* is a non-ground component. This is due to the fact that such a literal does not correspond to any component.

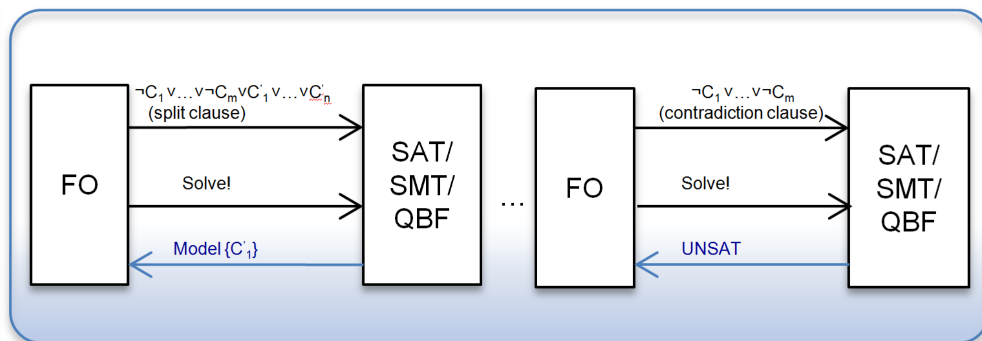

Figure 5.1: Interaction between the SAT solver and the FO reasoning part in AVATAR.

In our context a SAT solver has to expose an incremental behavior. By incremental we mean that the solver receives from time to time new clauses that have to be added at the propositional problem and checks whether the problem is satisfiable upon request from the first-order reasoner. If the problem is satisfiable than all it has to do is to pass back to the first-order reasoner a model (component interpretation) for all the propositional variables. Otherwise it simply has to return unsatisfiable and communicate the unsatisfiability result to the first-order reasoning part as well.

5.1.2 AVATAR Algorithm

To better explain the cooperation between the SAT solver and the first-order reasoner, we have to modify the superposition calculus to deal with *assertions*. In the context of AVATAR the notion of an *assertion* is similar to the one used in splitting without backtracking but not entirely the same. In our context, we define an *assertion* to be a finite set of components.

We define a *clause with assertions* (*A-clause*) as being a pair that consists of a clause *X* and an assertion *A*. In general we denote an assertion clause by $(X \leftarrow A)$ or in case the assertion $A = \emptyset$ we denote the clause by X. Or more formally we an *A-clause* can be seen as $(X \leftarrow C_1, \ldots, C_m)$ which is equivalent to $\forall X \vee \neg\forall C_1 \vee \ldots \vee \neg\forall C_m$. Using this definition we can safely say that any standard clause X can be considered an *A-clause* that contains the empty set of assertions.

Since we are using a mapping from first-order to SAT we have to extend the mapping such that it works with clauses with assertions. The extension is straight forward, considering the same mapping function ([.]) as before we can extend it for an assertion as follows. Consider you have an assertion $A = \{C_1, \ldots, C_m\}$, we define $[A] = \{[C_1], \ldots, [C_m]\}$ to be the mapping of assertion A from FO to propositional.

Since AVATAR works with A-clauses instead of normal clauses we have to modify the superposition calculus to a calculus that uses A-clauses instead of standard clauses. This modification is done by transforming any inference rule for the superposition calculus that has the form:

$$\frac{X_1 \qquad \ldots \qquad X_k}{X}$$

into a set of rules of the form :

$$\frac{(X \leftarrow A_1) \qquad \ldots \qquad (X_k \leftarrow A_k)}{(X \leftarrow A_1 \cup \ldots \cup A_k)}$$

where $A_1, \ldots, A_k$ are assertions. Also keep in mind that at each point in time AVATAR uses as assertions those components that are computed by the SAT solver in its last model. The problem that we have to overcome now is that of changing SAT models, that translates into clauses with assertions being added or deleted at any point in time.

In order to cope with the fact that clauses with assertions can be deleted and in the next step undeleted the notion of *locked* A-clause. Throughout the modified saturation algorithm this kind of storage will be denoted by *locked*. Elements of this *locked* storage are pairs of the form (F, λ) where F is an A-clause and λ, also called lock of F, is a set of component literals (C-literals). Note that in this case an A-clause F can occur multiple times in *locked* by having different locks.

We say that an interpretation I returned by the SAT solver *unlocks* a pair $((X \leftarrow A), \lambda)$ if all the C-literals in $[A]$ are true in the interpretation and at least one C-literal in λ is either false or undefined in A. Throughout the execution AVATAR maintains the following invariant: for every A-clause $(D \leftarrow A)$ in the search space each of the literals in $[A]$ is true in this model. That is, for all the pairs (F, λ) that are added to the set of *locked*, all of the literals in λ are true in the current model, *interp*. If throughout the computation one of the literals in λ become false or undefined, then the pair (F, λ) has to be removed from the *locked* and added to the set of unprocessed clauses, lines 20-21 from Algorithm 6.

Notice that in the case of AVATAR algorithm we have to keep track of more collections than a general saturation algorithm that keeps track only of *active, passive* and *unprocessed*. In our case we have to keep track of the model produced by the SAT solver, that is a C-interpretation, deprecated in the algorithm by *interp*. This interpretation is needed so that at each point we can keep track of which clauses get locked or unlocked. In order for this operation to be performed the difference between previous C-interpretation and the current one is done.

Besides the interpretation one also needs to keep track of the clauses that have to be passed to the SAT solver. We do this by maintaining the *sat_queue* and adding to it clauses for the SAT solver. One can avoid keeping track of this queue by simply adding the SAT clauses to the solver as they are created. This raises the problem of computation of locked and unlocked clauses after every new clause is added to the SAT solver slowing down the prover, since in most of the cases the interpretations change. The only exception appears in case an empty A-clause is derived. In this case we directly add it to the SAT solver and recompute the interpretation. This is due to the fact that the new interpretation will make the given clause, and potentially many other, locked or even deleted.

Of course these changes influence the way simplifation and term indexing works. Details about how this changes influence the subsumptions, subsumption resolution and rewriting by unit equalities (demodulation) can be found in AVATAR paper [103]. In the same paper details about how indexing works in the context of AVATAR is explained.

SAT Solvers for AVATAR

In the context of AVATAR a SAT solver has to behave in the way they are designed for. Baiscally one does not have to modify the SAT solver in order to integrate it in

Algorithm 6 Modified version of the first-order algorithm

```
1: Var active, passive, unprocessed: set of clauses;
2: Var given, new: A-clause;
3: Var sat_queue: set of A-clauses
4: Var locked: set of pairs (A-clause, lock)
5: Var interp: C-interpretation
6: for all D ∈ initial clauses do
7:     if D is splittable or empty then
8:         move it to sat_queue
9:     else
10:        move it to unprocessed
11: main loop
12: if sat_queue ≠ ∅ then
13:     for all A-clauses (C_1 ∨ ... ∨ C_n ← C'_1 ∨ ... ∨ C'_m) ∈ sat_queue do
14:         pass [C_1] ∨ ... ∨ [C_n] ∨ ¬[C'_1] ∨ ... ∨ ¬[C'_m] to SAT-solver
15:     sat_queue := ∅
16:     Ask SAT-solver to solve the problem
17:     if SAT-solver returns unsatisfiable then
18:         Return unsatisfiable
19:     interp := C-interpretation returned by SAT-solver
20:     for all pairs((C ← A), λ) ∈ locked unlocked by interp do
21:         remove the pair from locked and add (C ← A) to unprocessed
22:     for all A-clauses (C ← A) ∈active,passive or unprocessed s.t [A] ⊈ interp do
23:         remove (C ← A) from the set and add ((C ← A), ∅) to locked
24:     for all components [C] ∈ interp s.t (C ← C) ∉ active ∪ passive ∪ unprocessed do
25:         add (C ← C) to unprocessed
26:     for all new ∈ unprocessed do
27:         if new is splittable or empty then
28:             add new to sat_queue
29:         else
30:             if retained(new) then
31:                 simplify new by clauses in active ∪ passive
32:                 if new is added to unprocessed then
33:                     simplify clauses in active ∪ passive by new
34: if sat_queue is non-empty then
35:     start the main loop again
36: if passive = ∅ then
37:     Return satisfiable or unknown
38: given := select(passive)
39: move given from passive to active
40: unprocessed := forward_infer(given, active)
41: add backward_infer(given, active) to unprocessed
```

this architecture. The only desired behavior that should be exposed by the SAT solver would be incrementallity. In the following we will give some details about the SAT solver implemented in Vampire, in the context of AVATAR architecture.

Vampire Default SAT Solver

Vampire implements it's own SAT solver that follows the MiniSAT [42] guidelines, thus implementing a variation of CDCL architecture. That is, it implements an incremental SAT solver that uses most of the optimizations presented in the context of MiniSAT. The advantage of havins a SAT solver implemented in the context of Vampire is that one can easily customize the behavior of such a component to fit different needs. In our case, assuming that the SAT solver returns that the problem is unsatisfiable, then also the FO reasoner should report unsatisfiability of the problem.

The problem that arises here consists on giving a proof of unsatisfiability. Since we use a SAT solver in the background it is the solver's job to prove a trace of how unsatisfiability was decided and then ask the FO reasoning part to output the FO proof based on the proof of unsatisfiability returned by the SAT solver.

Of course, one can argue that outputing the refutation proof based on the entire SAT problem actually gives a good proof for refutation. But in case we are interested in having succint and readable proofs one would need to select only those clauses that where involved in proving unsatisfiability of the SAT problem. Such a behavior is actually implemented in the case of the Vampire SAT solver.

Lingeling

Lingeling is the best performing SAT solver over the last years SAT competition. The solver, implements a variation of the CDCL algorithm and a multitude of inprocessing and preprocessing techniques. Although it is used in commercial applications, the source code of Lingeling is open for academic and non comercial use.

Details about different preprocessing techniques implemented by Lingeling can be found in [41]. More details about Lingeling's interface can be found in [16], where PicoSAT's interface is presented. In the case of Lingeling, the interface has similar functionality as the one of PicoSAT. Also for a in-depth description of the inprocessing rules, we reffer to [58].

5.2 Integrating Lingeling SAT Solver in AVATAR

We now describe how we integrated the Lingeling SAT solver in the framework of Vampire. When we describe implementation decisions also an overview of the options implemented in order to control the behavior of Lingeling in the AVATAR framework of Vampire. Although Lingeling is used in commercial applications, the source code

is publicly available. Also the default license allows Lingeling to be used in non-commercial and academic context. Our main goal after integrating the new solver in Vampire framework was to obtain better performance in the process of solving first-order problems.

5.2.1 Integrating Lingeling

In general any SAT solver is designed to accept as its input problems described in the DIMACS format [16]. We have decided to implement an interface that allows us to directly control Lingeling via its API. By using the API one can control the options for the background SAT solver much easier at run time depending on the strategy being deployed. We call a *strategy* a combination of options that are meant to influence the behavior of different components of Vampire.

As presented in the AVATAR algorithm, in case the SAT solver establishes satisfiability of a given problem, we are interested in obtaining a model for the problem. This model is basically equivalent to the *interp*, C-interpretation, from Algorithm 6. Since this is all the interaction between the FO reasoner and the SAT solver in case of satisfiability, we can safely say that it matches the intended use for the majority of SAT solvers on the market, in particular also Lingeling's.

Many computation steps depend on the model generated by the SAT solver, take for example indexing, forward simplification steps. In this particular case we are also interested in finding similar models upon two different calls to the SAT solver. That is, in case the SAT solver returns satisfiable, and we add new clauses to the solver and the status of the problem remains satisfiable, we are interested in a model that has minimal change when compared to the previous one. *Similar models* reffer to those models that have small number of changes (ideally no change to the previous model and only assignments to the newly added SAT variables) when two or multiple times the SAT solver is called on the problem in incremental way. As mentioned before, by doing so we are actually minimizing the work that has to be done by the FO reasoning part after each call to the SAT solver.

For the purpose of our work, we use Lingeling in an incremental manner, but there is still the question of how should we add the clauses to the solver. Incrementality in the context of SAT solving refers to the fact that a SAT solver is expected to be invoked multiple times. Each time it is asked to check satisfiability status of all the available clauses under assumptions that hold only at that specific invocation. The problem to be solved thus grows upon each call to add new clauses to the solver, for details see [42].

In the context of Vampire at some particular point the first-order reasoner can add a set of clauses to the existing problem. In order to add these clauses to the underlying SAT solver we implemented two versions of using Lingeling in the AVATAR architecture of Vampire. The first version, given in Algorithm 7, iterates over the clauses that appear in the original problem and adds them one by one to Lingeling. After we have

added the entire set of clauses to the SAT solver we call for satisfiability check. We call this method of adding clauses "almost incremental" since it does not call for satisfiability check after each clause is added. Algorithm 7 is very similar to non-incremental SAT solving at each step when the first-order reasoning part asks for satisfiability check, since the call for satisfiability is done only after all the new clauses are added to the solver (line 5). Overall, the approach is still based on incrementality of the underlying SAT solver, since we keep adding clauses to the initial problem.

Algorithm 7 *"Almost" incremental* version of Lingeling in Vampire

1: **Input:** a set of clauses to be added
2: **while** not all clauses added **do**
3:     **Add** clause to Lingeling
4:     **Keep track** of the added clause
5: **Call** SAT procedure
6: **if** UNSATISFIABLE found **then**
7:     **Report** Unsatisfiability
8: **else**
9:     **Return** a model

Another way of using the underlying solver would be to simulate the pure incremental approach, as presented in Algorithm 8. This approach is similar to the previous one with the difference that now as soon as a new clause is added to Lingeling we are also calling for a satisfiability check (line 4).

In order to be able to use any of the previous ways of integrating Lingeling in Vampire one has to be careful when adding clauses to Lingeling. Internally Lingeling tries to apply preprocessing on the problem and during preprocessing a subset of variables could be eliminated. This can lead to some problems since we eliminate a subset variables during the preprocessing and in some future step we might add some of them back to the solver. The issue that arises here is the fact that performing these operations can lead to unsoundness of the splitting solution generated by the first-order reasoner.

In order to avoid the issue of not allowing the solver to eliminate variables while performing the preprocessing steps, Lingeling relies on the notion of *frozen* literals [17]. One can see a frozen literal as a literal that is marked as being important and not allowing the preprocessor to eliminate it during preprocessing steps. Using freezing of literals we are ensured that although preprocessing steps are done, it will inhibit the elimination of marked variables. In our case it actually means that one has to freeze all the literals that appear in the initial problem and also all the literals that are due to be added. The process of freezing literals is done on the fly when new clauses are added to the solver. In order to do be efficient and not freeze multiple times the same literal we keep a list of previously added and frozen literals.

Algorithm 8 *Incremental* version of Lingeling in Vampire

1: **Input:** a set of clauses to be added
2: **while** not all clauses added **do**
3:     **Add** clause to Lingeling
4:     **Call** SAT procedure
5:     **Keep track** of the added clause
6:     **if** UNSATISFIABLE found **then**
7:         **Report** Unsatisfiability
8:     **else**
9:         **Return** a model

Although the freezing of all literals proves to be a suitable solution of enforcing Lingeling not to eliminate some variables during the preprocessing steps, this also limits the power of the preprocessing implemented in the solver. One improvement could be to develop a methodology that would allow "predicting" which literals are not going to be used later on and allow the SAT solver to eliminate them if necessary.

5.2.2 Using Lingeling in Vampire

In order to run Vampire[1] with Lingeling as a background SAT solver one has to use from command line the following option:

```
-sat_solver  lingeling
```

By default when one enables the use of Lingeling as a background SAT solver, the solver is used as presented in Algorithm 7. This means that we add first all the clauses to the SAT solver and only then call for satisfiability check.

In case one wants to use Lingeling in Vampire as presented in Algorithm 8 the following option needs to be used

```
-sat_lingeling_incremental  [on/off]
```

This enables the incremental use of Lingeling as presented in the algorithm. By default this option is set to **off**.

We are also interested in generating similar models when we use incrementally the underlying solver. In order to control this behavior, one should use the option:

```
-sat_similar_models  [on/off]
```

By default this option is set to **off**. As for the previous options activating similar model generation has effect only in the case where Lingeling is used as background solver. In the following we present the results obtained by running Vampire with combinations of these options.

[1]Vampire with all the features presented in this paper can be downloaded from `vprover.org`

Strategy	Vamp	L	L S	L I	L I S
Average Time	3.4747	3.0483	4.2159	2.6728	3.8490
# of solved instances	142	146	156	143	144
# different	2	12	16	10	11

Table 5.1: Results of running Vampire with default values for parameters on the 300 CASC problems.

5.3 Experimenting with Vampire's AVATAR

Currently, there are all together 5 different combinations of values for the new options controlling the use of SAT solvers in Vampire. In order to benchmark these strategies we used problems coming from the TPTP [102] library. The experiments where run on the InfraGrid infrastructure of West University of Timisoara [5]. The infrastructure contains 100 Intel Quad Core processors, each one with dedicated 10GB of RAM. All the experiments presented in this paper are run with a time limit of 60 seconds and with memory limit of 2GB.

5.3.1 Benchmarks and Results

As a first set of problems we have considered the 300 problems from the first-order division of the CASC 2013 competition see [101]. Besides these problems we also used 6637 problems from the TPTP library. These 6637 problems are a subset of the TPTP library that have ranking greater than 0.2 and less than 1. Ranking 0.2 means that 80% of the state-of-the-art automatic theorem provers can solve this problem, while ranking 1 means that no state-of-the-art automated theorem prover can solve the problem.

Generally using a cocktail of strategies on a single problem proves to behave always better in first-order automated theorem proving. For this purpose we have decided to evaluate our approaches of using SAT solving in the AVATAR framework of Vampire both using a mixture of options and also using the default options implemented in Vampire.

CASC Competition Problems

We evaluated Vampire using all the new SAT features and kept all other options with their default values, from now on we will call this version of Vampire *default mode*. Also we evaluated the mode where we launch a cocktail of options (strategies) with small time limits and try to solve the problem, called the *casc mode*. A summary of the results obtained by running these strategies can be found in Table 5.1 and Table 5.2.

Strategy	Vamp	L	L S	L I	L I S
Average Time	3.4679	3.0615	4.2701	2.8139	3.7852
# of solved instances	230	233	240	232	232
# different	1	8	13	8	7

Table 5.2: Results of running Vampire using a cocktail of strategies on the 300 CASC problems.

All tables presented in this paper follow the same structure: the first row presents the abbreviations for all the used strategies, the second row presents the average time used by each of the strategies in order for solving the problems. Here we take into account only the time spent on the problems that can be solved using a particular strategy. The third row presents the total number of problems solved by each strategy. The last row presents the number of different problems.

By different problems we mean problems that could be solved either by Vampire with the default SAT solver and not solved by any of the strategies involving Lingeling and the problems that can be solved only by at least one strategy that involves Lingeling but cannot be solved by Vampire using the default SAT solver.

The abbreviations that appear in the header of each table stand for the following: *vamp* stands for Vampire using the default SAT solver, *L* stands for Vampire using Lingeling as background SAT solver, in an "almost" incremental way, *L S* similar to *L* but turning the generation of similar models on the SAT solver side on, *L I* stands for Vampire using Lingeling as background SAT solver in pure incremental way and *L I S* is similar to *L I* but with the change that it turns similar model generation on the SAT solver side.

Table 5.1 reports on our experiments using the default mode of Vampire on the 300 CASC problems. Among these 300 problems, 23 problems can be solved only by either Vampire using some variations of Lingeling as background SAT solver or by Vampire using the default SAT solver. Table 5.2 shows our results obtained by running Vampire in casc mode on the 300 CASC problems. Among these 300 problems there are 18 problems that can be solved only by either Vampire using some variation of Lingeling as background SAT solver or by Vampire using the default SAT solver.

Figure 5.2 presents a comparison between Vampire using the default SAT solver and each of the new strategies. The scatter plots present on the x-axis the time spent by Vampire in trying to solve an instance, while on the y-axis the time spent by different strategies on the same instance. In order to have more concise figures we have decided to normalize the time spent in solving by a factor of 10. Doing so one can compare time-wise the performance of each of the strategies. A point appearing on the diagonal of the plot represents the fact that both strategies terminated in the same amount of time.

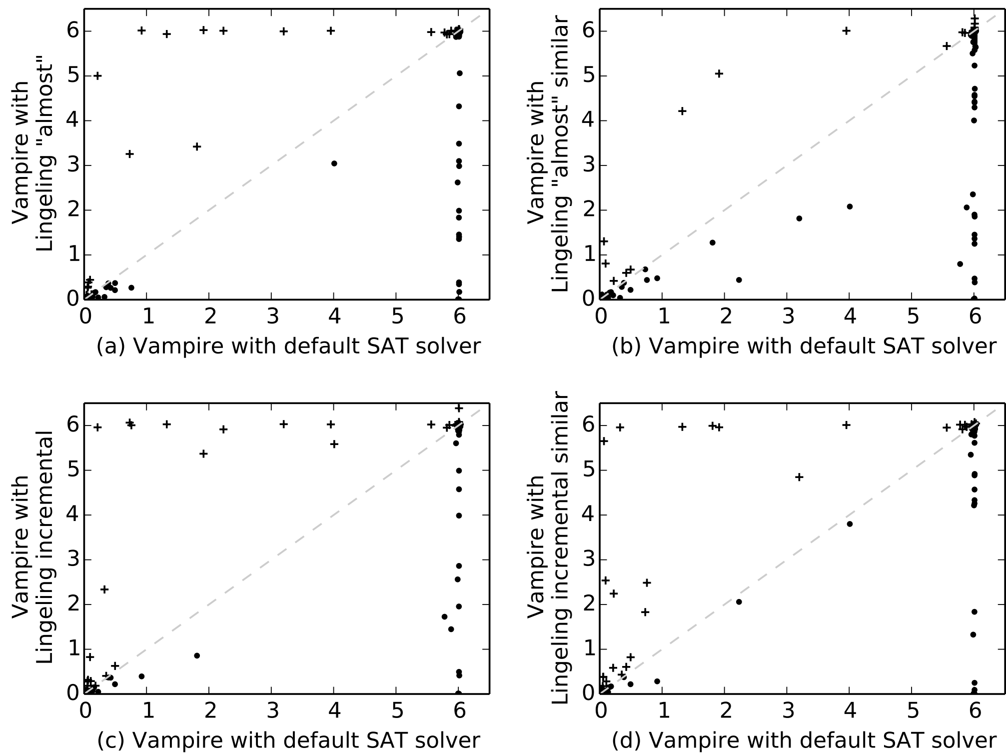

Figure 5.2: Comparison of performance between Vampire with the default SAT solver and different Lingeling strategies run in *default* mode. Default SAT solver is compared against: (a) Lingeling "almost" incremental, (b) Lingeling "almost" incremental and similar models, (c) Lingeling incremental and (d) Lingeling incremental and similar models

A point appearing on top of the main diagonal represents the fact that Vampire using the default strategy managed to solve that instance faster than Vampire using Lingeling variations. Similar a point below the diagonal represents the fact that Vampire using the new strategy solved the problem faster than Vampire using the default SAT solver.

The plot presents one to one comparison between the strategies and the default strategy. In Figure 5.2 we present the results obtained by running Vampire in "default" mode but varying the SAT solver as described above. From this figure one can notice the fact that more points appear above the diagonal, meaning that the default values of Vampire are better. We can notice however that there are some problems on which Vampire with default SAT solver time out while using Lingeling they can be solved in very short time. From these plots one could conclude that taken individually these strategies and compared to the default one, they seem to be have similar behavior as the default one.

Nevertheless, if we take them together and compare them to the default strategy we notice the fact that indeed they behave better.

Table 5.2 and Figure 5.3 present a similar comparison on the same problems, using the same variations of the underlying SAT solver and the same limits as for the *default* mode.

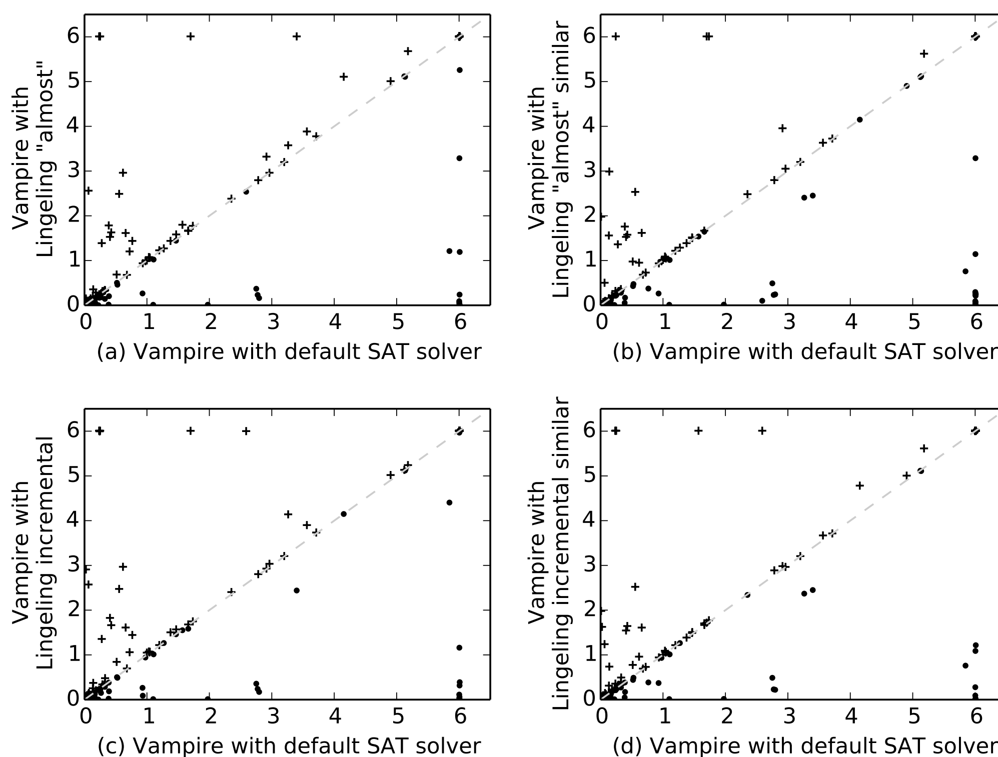

Figure 5.3: Comparison of performance between Vampire with the Default SAT solver and different Lingeling strategies run in *casc* mode. Default SAT solver is compared against: (a) Lingeling “almost” incremental, (b) Lingeling “almost” incremental and similar models, (c) Lingeling incremental and (d) Lingeling incremental and similar models

Other TPTP Problems

In a similar manner as for the 300 CASC problems we have evaluated our newly added features on a big subset of TPTP problems. The problems that have been selected for test have ranking in the interval [0.2, 1), having the status of either: *Unsatisfiable, Open, Theorem* or *Unknown*.

Strategy	Vamp	L	L S	L I	L I S
Average Time	6.0440	5.9982	6.5992	5.6805	6.5025
# of solved instances	2672	2810	2925	2750	2788
# different	104	350	422	328	334

Table 5.3: Results of running Vampire using default values for parameters on the 6.5K problems.

Strategy	Vamp	L	L S	L I	L I S
Average Time	6.1019	6.0895	6.1139	6.3069	6.0638
# of solved instances	4788	4822	4881	4809	4792
# different	81	212	245	207	194

Table 5.4: Results of running Vampire using a cocktail of strategies on the 6.5K problems.

In Table 5.3 we present the summry of obtained results from running Vampire with all the variations on the set of problems in *default* mode. Table 5.4 presents the summary of our results obtained by running Vampire in *casc* mode on the same set of problems using the same variations as above described.

From our experiments we noticed that using Lingeling as a background SAT solver in the “almost” incremental and with the similar model generation turned on proves to perform the best among the newly implemented strategies. This sort of behavior can be due to multiple reasons. First it could be due to the fact that the solver tries to keep the model for as long as possible, due to similar model generation option. Another explanation for best performance can be the fact that using this options we do not call the SAT solver after each clause is added, but rather only after we add all the clauses generated by the first-order reasoning part, hence decreasing the time spent by the SAT solver in solving.

5.3.2 Analysis of Experimental Results

While integrating the new SAT solver inside Vampire and during the experiments we observed some issues that might increase the performance of future SAT solvers inside the Vampire’s AVATAR architecture.

(i) It is not necessary that a state-of-the-art SAT solver, as Lingeling, behaves better inside the AVATAR framework.

(ii) Integration of new solvers is less complicated than fine tuning the newly integrated solvers in order to match the performance of the default SAT solver, which is hard.

(iii) Using an external SAT solver just in case the SAT problems are hard enough could be a good trade-off. We discuss these issues below.

Performance First, let us discuss the performance issue. At least in the case of Lingeling upon integration we have noticed that it behaves really nice on some of the problems while on some others it seems to fail. There are a couple of factors that could influence the behavior of such a performant tool. (1) Calling the solver many times decreases its performance. Although the solver is designed to be incremental upon adding new clauses to the problem and call for satisfiability check, it restarts. Now the problem appears when we call many times the solver. For example in the case of "pure" incremental way, we call the solver after each clause is added. Although the speed with which the check is done is incredibly fast, calling it n times makes it n times slower. (2) Due to this behavior in the worst case we have n restarts on the SAT solver side, where n is the number of clauses added to the solver. Both these points showed up in the statistics from the experiments we have performed. It is not uncommon that the first-order reasoning part will create a problem containing 10K or even 100K clauses. Now even if for one call the SAT solver spends 0.01 seconds it results in a timeout.

Own SAT solver? The default SAT solver implemented in Vampire follows the general structure of the MiniSAT [42] SAT solver. This architecture is an instantiation of the Conflict-Driven Clause Learning (CDCL) [97] architecture. Although it is incremental, it deals with incrementality in a different manner. Assuming that we add a clause to the solver, first we check whether we can extend the current model so that we can satisfy also the newly added clause if the clause gets satisfied by the current model we keep the model and add the clause to the database. In case the clause is not directly satisfied by the model but does not contain any variable that is used in the model we try to satisfy the clause by extending the model. If we cannot do that, we do not restart, but rather backtrack to the point where the conflict comes from. A conflict can appear only if variables that are used in the model appear also in the newly added clause. If that is the case we take the lowest decision level among the conflict variables and backtrack to it. From there on we continue the classical SAT procedure and try to find a new model.

Using this approach, the SAT solver brings a couple of advantages for the first-order reasoning part. Due to the fact that we try to keep the model with minimal changes, we do not have to modify the indexing structures so often in the first-order part and also from our experiments we have noted that the actual SAT solving procedure gets called less often than in the case of calling for satisfiability check on an plugged-in solver.

Integrating other solvers Adding a new external SAT solver inside the framework of Vampire is not complicated. One has to take care about how the first-order reasoning part and the SAT solver communicate and do some book keeping. The issue arises when one has to fine-tune the SAT solver so that it performs well in it's new environment. Usually state-of-the-art SAT solvers are highly optimized in order to behave well on big problems coming from industry but sometimes seem to get stuck in small problems. Also one important component of a state-of-the-art solver is preprocessing [41], which for our purpose has to be turned off in order to ensure that we do not eliminate variables that might be used for splitting in the first-order reasoner. Due to the fact that we do not use all the power of a SAT solver we have to answer the question whether it is the case that state-of-the-art, or commercial, solvers behave better in this context. In the case of the AVATAR architecture for an automated theorem prover producing different models for the SAT problems means that a clause gets splitted in different ways. That also translates into the fact that in some cases the use of an "handcrafted" SAT solver might produce the right model, but it also means that in other cases the state-of-the-art solver produces the right model at the right time. This results in either solving the problem really fast or not at all. Some interesting fact that we have noticed during the experiments is the fact that the AVATAR architecture is really sensitive towards the models produced by the SAT solver.

Future Directions

Starting from the initial results of the newly introduced AVATAR framework for an automatic theorem prover, we investigate how does a state-of-the-art SAT solver behave in this framework. We describe the process of integrating a new SAT solver in the framework of Vampire using the AVATAR architecture. We also present a couple of decisions that have been made in order to better integrate the first-order proving part of Vampire with SAT solving. From our experiments we noticed that using a state-of-the-art SAT solver like Lingeling inside the framework of an automated theorem prover based on AVATAR is useful and behaves well on TPTP problems. However there are also cases where using Lingeling as background solver Vampire does not perform as good as using a less efficient SAT solver.

We believe that further refinements on the SAT solver part and better fine tuning of the solver will produce even better results. We are investigating different ways of combining the Vampire built-in SAT solver with the external SAT solver such that we do not restart upon every newly added clause. Besides splitting, Vampire uses SAT solving also for instance generation [69] and indexing. We are therefore also interested in finding out whether the use of a state-of-the-art SAT solver improves the performance of Vampire.

CHAPTER 6

Related Work

Verifying program properties using a first-order automatic theorem prover is challenging and actively studied reseach problem. Besides verification also the problem of generating properties about programs poses lots of interesting questions that are still waiting for an answer. In the following we make a short overview of different approaches that are described in literature and which are directly related to the research directions carried out throughout this thesis.

6.1 Invariant Generation

To the best of our knowledge symbol elimination method for generating loop invariants for programs containing arrays is implemented only in the context of Vampire. The way invariants are generated is different than the other presented methods in multiple ways. Symbol elimination does not need any user guidance and the generation of invariants is based on a modified version of Vampire, saturation based first-order theorem prover. On the other hand some other state-of-the-art approaches prove to produce really good results as well.

McMillan in [79] proposes a different way of generating universally quantified loop invariants. The method proposed here is based on modifying an interpolating theorem prover to generate invariants. In order to achieve this goal the theorem prover has to be enhanced with so called procrastination inference rules. After doing so one has to ensure that the refutation proofs that are generated by the prover are local and only then the the proof can be used in order to generate invariants. Besides locality of the proofs one also has to make sure that the proofs do not diverge. This work also proves to be a starting point in the study of controlling divergence of proofs. In order to do so their method is based on successive loop unrolling until an inductive invariant is found. Although the

method can be fully automated it is a bit more complicated to fully grasp. In this work the authors present also the modifications done to SPASS [105] theorem prover in order to obtain invariants for some simple programs that handle arrays and lists. Although the generated invariants prove to be useful in small practical examples using this type of invariant generation does not generate invariants with quantifier alternations, thus making the invariants harder for a human reader to understand.

Unlike the method proposed by McMillan, Srivastava and Gulwani in [99] propose a different approach for generating quantified invariants. That is, they propose a method not completely automatic but rather a semi-automatic process where they use an SMT solver as a black box. This method is based on the concept of invariant templates provided by the programmer. These templates contain "holes" that must be filled by the analysis. Although this technique is not fully automatic it benefits of the advances in SMT solvers, since it makes use of them in order to fill the invariants. The downside of using such a method would be the fact that users, programmers, have to annotate the code with templates for invariants. On the plus side however, using the techniques proposed by the authors one can verify more complex programs, that is it can safely analyze and prove intended properties of nested loops. Whereas the symbol elimination method, although fully automatic still cannot achieve that.

In [14] the authors propose a modification of the counter-example guided abstraction refinement technique (CEGAR) [23] for program verification. The modifications that are added to this technique aim to limit the number of false positives that are generated. For this purpose the so called "path invariants" are used. These invariants are basically template based inducted invariants generated for parts of the original program. To be more precise, that part of the program that is responsible for generation of the spurious counterexample in the CEGAR loop. In order to generate path invariants the authors make use of the technique presented in [15]. This technique is also part of the class of techniques that use templates in order to generate invariants but in the combined theory of linear arithmetic and uninterpreted functions. Unlike the method proposed by Gulwani where an SMT solver is used, in the approach proposed here invariant synthesis is done by constraint solving.

All the above methods make use of both properties to prove and/or templates, while in the case of symbol elimination, we do not need properties to prove and/or templates in order to generate properties for arrays. Similarly in the sense that no user intervention is needed, Halbwachs et al. in [52] present a way of generating properties about array content. The method presented in this paper, is inspired from the one presented in [49] that is, each array gets mapped into symbolic intervals. For each such interval and for each array a new variable is considered and relational properties about these variables are taken into consideration.

In [52], the semantics of simple programs mapped to intervals is described as well as a first implementation of the method and initial experimental results. In [85] for

proving more interesting properties about content of arrays the authors deploy a different approach than the one of mapping arrays to sets. More precise they are interested in properties that cannot be expressed by simple quantifier alternation in first-order logic, like the fact that two arrays are equal up to a permutation. In order to discover these global properties over arrays content, they use multisets and the method proposed is basically an abstract interpretation that propagates different multiset equations.

6.2 Arithmetic Reasoning

In general when we speak about solving systems of linear inequalities we have methods like Fourier-Motzkin variable elimination and Simplex method plus their variations. Following another style of reasoning we have methods that adopt the DPLL style reasoning for deciding satisfiability of a set of inequalities.

In the case of Fourier-Motzkin [29, 93] variable elimination the method works as follows. Given a system of linear inequalities we try to decide satisfiability of this system by repetitive transformations of the system. Transformations are nothing else but rewriting of the system in such a way that after each iteration a variable gets eliminated and the set of solutions for both system are the same over the remaining variables. In case all the variables are eliminated from the system of linear inequalities, then one obtains a system of constant inequalities. As a consequence, elimination of all variables can be used to detect whether a system of inequalities has solutions or not. That is, in case the final system contains $\bot$ then the initial system of inequalities is unsatisfiable. Otherwise it is straight forward to construct a solution for the initial system of inequalities. Note that by applying this algorithm we are assured termination and only a finite number of new inequalities are generated while running. Another important issue that a predefined ordering on the variables has to be set from the beginning. This is actually not the case for bound propagation, where variable ordering is not fixed.

The Simplex [20] method is one of the oldest methods used in order to solve linear optimization problems, that is to solve a system of linear inequalities and minimize/maximize an objective function. Besides this intended goal, variants of the simplex algorithm can be used in order to solve also the decision problem for quantifier-free fragment of linear arithmetic. The algorithm works iteratively and finds feasible solutions satisfying the given constraints and greedily tries to minimize/maximize the objective function. In the case of a the decision problem, the algorithm focuses on finding a single feasible solution that satisfies all the constraints. A variant of the Simplex algorithm, called *dual Simplex* works in a similar manner but is really effective in the case when constraints are added in an incremental way.

The big step forward towards modern implementations of Simplex inside of SMT solvers is considered to be the method introduced in [40] by Dutertre and de Moura. They propose a new way of integrating Simplex in the context of DPLL(T). Where the

major advantage would be that by using this method one can benefit of all the advantages of the DPLL(T) framework.

Conflict resolution is a method recently introduced in [64] and its first implementation is evaluated in [65]. As per all the new style methods, the conflict resolution method is an iterative and solution driven method. This method starts from a arbitrary initial assignment for all the variables that are present in the system to solve and iteratively tries to refine these values. During the refinement process, either a solution for the system is found, or a conflict arises. Conflicts are basically pairs of inequalities that do not allow the refinement of current assignments to variables to satisfy the system of inequalities. In this case the conflict is resolved by deriving a new inequality from the two conflicting inequalities and it is added to the original system. The process of refining the assignment continues until either the assignment is refined into a solution or a trivial inequality is derived. In the latter case it is safe to conclude that the initial system is unsatisfiable.

Now more to the second class of algorithms, McMillan proposes a new DPLL procedure in [80], called generalized DPLL method , shortly called GDPLL. The newly introduced method exploits some ideas from DPLL to reason in the quantifier-free linear real arithmetic. This method tries to find an assignment satisfying a given system of linear inequalities. For doing so, values to variables are assigned in an iterative process. As in the case of DPLL reasoning in case a conflict is detected a new clause is learned. After this new conflict clause is leaned it further used in order to resolve conflicts. The theory reasoning part of GDPLL is essentially equivalent to the conflict resolution method [64]. As in the case of conflict resolution, when computing and assigning values of variables an a priori fixed variable ordering is used. The efficiency of conflict resolution crucially depends on the chosen ordering. The inability to change the ordering during the proof search is probably the main weakness of the method.

Effects of variable ordering are also studied in the framework of algebraic computation [21, 37]. Although the work presented in this framework addresses a different problem than ours, it is still interesting since it shows potential improvement given good variable orderings.

Compared to the previous methods, bound propagation algorithm has the big advantage of incorporating different techniques from the SAT solving community. In the case of bound propagation algorithm, bound propagation technique is incorporated and acts like the unit propagation thinker from SAT community. Other features adapted from the SAT solving community include dynamic variable ordering, technique that proves to be essential in order to achieve good performance. Lemma learning and backjumping also significantly improve performance of the bound propagation algorithm.

While dynamic selection of variables gives bound propagation algorithm its flexibility, uncontrolled variable selection coupled with bound propagation can easily yield to a non-terminating algorithm. Therefore termination is a highly non-trivial issue. In [67]

it is shown that under a natural restriction on bound propagation, the bound propagation algorithm always terminates. Whereas if the variable ordering is changing termination is not guaranteed anymore for the GDPLL or in the case of conflict resolution.

Dynamic variable selection is also used in [25], by combining theory reasoning with a DPLL-style algorithm. The method implements the variable state independent decaying sum (VSIDS) heuristic of [81] for variable ordering. Unlike BPA, completeness of this method is guaranteed only under some restrictions, for example assuming that the chain of resolving steps is finite. Another major difference to bound propagation algorithm is the fact that bounds are not propagated, while bound propagation is one of the key features of the BPA algorithm.

A somewhat similar approach to the one proposed by bound propagation algorithm is described in [60] in the context of integer linear programming. The key difference is that [60] does not use collapsing equalities. In general collapsing inequalities are essential for turning a choice among an infinite number of values for a numeric variable in a given interval into a non-deterministic don't care selection of a value in this interval.

As evidenced by the above described approaches, reasoning about linear arithmetic crucially depends on the underlying theory and heuristics for variable orderings and clause learning. BPA has been introduced only recently in [67] and its practical efficiency is yet unknown. In [39] we present the first ever implementation of the method. Also the paper undertakes the first investigation into understanding the power and limitations of BPA for linear real arithmetic. To this end, we implemented and studied various heuristics for variable orderings, assigning values to variables and learning new inequalities.

6.3 Reasoning with Theories

In the latest years the problem of reasoning with combination of theories has attracted lots of attention. Following the advances in the SAT community the field of reasoning with combinations of theories has advanced significantly. In the latter case tools like Yice [40], CVC [10, 11] and Z3 [33] appeared.

Tools like the aforementioned ones are efficient when it comes to reasoning with ground instances and with combination of theories. The architecture of such a tool is in general an instantiation of the DPLL(T) architecture, more details about DPLL(T) for SMT can be found in [34, 40]. In general in order to develop an efficient SMT solver one has to integrate a performant SAT solver and to tightly couple theory reasoning with SAT solving.

When it comes to program verification, in general it is desired to reason with both combinations of theories and with quantifiers. For this problem the approach used in SMT solvers is not so performant. First-order theorem provers are really efficient when

it comes to reasoning with quantifiers, but they lack efficiency when it comes to reasoning with combination of theories.

In order to tackle this problem two major directions are presented in literature. One direction is to start from the SMT community and extend SMT solvers in order to better handle quantifier reasoning and to integrate a saturation engine inside them, see [34]. The other direction would be to start from a first-order theorem prover and try to integrate different theory reasoning engines inside and tightly couple them with the saturation engine that is already implemented, see [103].

In the direction from SMT solvers towards first-order theorem provers with integrated theories we have the extension of the DPLL(T) and addition of a saturation engine. DPLL(T) is extended as follows. First the notion of *hypothesis* that keeps track of dependencies on case splits when the saturation applies inferences. In order to achieve this goal, the deduction rules deployed by the saturation engine are lifted in order to keep track of the hypothesis. By using the modified inference rules and the notion of hypothesis one can tightly couple the saturation engine and the DPLL component of the solver. One major advantage of using this technique is that it is refutationally complete, for more details about how this technique works see [34].

On the other direction, when coming from an full first-order theorem prover towards an SMT solver, we have the AVATAR architecture [103]. In a sense this architecture is similar to the one presented in [34] but with a couple essential differences. AVATAR architecture is flexible and easy to extend in order to integrate different engines that can be tightly coupled with the saturation engine. In order to achieve this, the AVATAR architecture makes use of so called *assertions* and one has to modify the inference rules so that assertions are propagated when the search advances. Assertions have the role of keeping track of the case splits that are done and are tightly coupled with the underlying SAT solver. A couple of problems that have to be overcome when implementing a solver using this architecture consists in keeping track of deleted clauses, since they can be undeleted later. One can find more details about how the saturation algorithm and the inferences have to be modified in order to integrate the AVATAR architecture in [103]. In AVATAR the SAT solver is used mainly for splitting a clause into components. Some of the advantages of using such an architecture is that one can easily integrate different back-end solvers, like SMT or quantified boolean solvers (QBF), instead of an SAT solver for reasoning with different theories.

As per the case of an SMT solver, first-order theorem provers that implement the AVATAR architecture rely on efficient background SAT solvers.

For the last couple of years SAT solvers have seen a great improvement in performance. Good performance for satisfiability solvers started to become common after the work of Stallman and Sussman [100] that introduced the notion of dependency-directed backtracking. More recent improvements come from the use of learning and use of non-chronological backtracking scheme, introduced in the mid 90's by Marques-Silva and

Sakallah and first presented in the context of GRASP [98].

Besides clause learning based on unit propagation, GRASP also proposed the unique implication points, becoming another hallmark in the SAT solvig community. Using the unique implication points, which are nothing else but dominators in the implication graph, one can better exploit the structure of unit propagation.

With the introduction of Chaff [81] a new technique for handling clauses was proposed, that is the *watched literals* scheme. Besides the watched literals, Chaff also introduced the VSIDS branching heuristic and the first unique implication point backtracking scheme [107].

Besides these classical techniques some recent techniques were proposed for improving performance of SAT solvers. For example simplification of the CNF formula proves to significantly improve the overall performance of the solver, see [41]. Also component caching and more complex clause learning schemes where proposed. Details about latest improvement in best performing SAT solvers can be found in entrants description for the SAT solving competition [13]. Another interesting optimization technique deals with restarts of a solver. It is empirically proven that by doing restarts at different points in the search chances of finding a solution increase, also overall performance of the solver increases. Other techniques including different preprocessing steps plus efficient data structures tailored for the purpose of SAT solvers have proven to make a big step forward for the SAT community.

CHAPTER 7

Conclusion

Formal verification is crucial for ensuring that software systems have no errors, and hence are correct. Automated methods are needed to analyze and formally verify computer systems. One challenge in the verification of software comes with the automated analysis of the logically complex part of code, such as loops or recursions. For such program parts, additional information about the program is needed in the form of auxiliary program properties. Typical program properties are loop invariants summarizing the loop behavior and describing safety properties of the program. Automated verification of programs with loops therefore crucially depends to which extend logically powerful loop invariants can be automatically generated.

In this thesis, we address the problem of automated invariant generation for program analysis and verification. Unlike existing approaches, in our work we propose the use of an automated first-order theorem prover not only for proving, but also for generating program properties. Our contribution are summarized below.

Invariant Generation. We use the recently introduced symbol elimination method in first-order theorem proving and experimentally study the power of symbol elimination for quantified invariant generation. To this end, our thesis provides an automated tool support, called Lingva, for generating loop invariants of programs with arrays. Lingva uses the award-winning first-order theorem prover Vampire and computes quantified invariants, possibly with quantifier alternations. The invariants generated by Lingva are obtained in a fully automated way, without any user guidance. We evaluated Lingva on a large set of academic and open-source benchmarks. Our experiments show that the invariants generated by Lingva capture the intended meaning of the program; in all examples we tried, user-given program annotations could be proven using our invariants.

Theory Reasoning. For proving program properties with both quantifiers and theories, we study the use of various decision procedures within a first-order theorem prover. In particular, we focus on linear arithmetic and provide an automated support for using Vampire for solving a system of linear inequalities over the reals or rationals. To this end, we integrate the bound propagation algorithm in Vampire. Our implementation provides a large number of strategies for choosing variable orderings and variable values, allowing us to experiment with the best options for variable and value selection within bound propagation. We evaluated our implementation on a large number of examples. Our experiments show that bound propagation in Vampire outperforms the best SMT solvers on some hard linear optimization problems.

SAT Solving. In order to go towards better integration of various decision procedures withing first-order theorem proving, we address the recently introduced AVATAR framework for combining automatic theorem proving with SAT/SMT reasoning. We describe the process of integrating different SAT solvers in the framework of Vampire using the AVATAR architecture, and experimentally investigate how the use of SAT solvers within Vampire improve overall performance of Vampire for proving first-order problems. To this end, we focus on the best state-of-the-art SAT solver, Lingeling and its integration and use within Vampire. We evaluated our implementation on the TPTP problem library of automated first-order theorem provers. From our experiments we note that using SAT solvers within Vampire allows us to prove problems that could not be proved so far.

Further Work. In the case of invariant generation an interesting topic for further research consists in extending our method in order to handle nested loops. Alongside with nested loops, we are also interested in finding ways of combining different theories, such as bitvector theory, datatypes, etc., that can be used for the purpose of invariant generation of more complex programs.

Another interesting topic for further research when it comes to bound propagation is improvement of the deployed strategies and thoroughly experiment with a large set of benchmarks. For better results one can develop fine tuned strategies for different classes of problems that are handled. Besides experimenting and strategy improvement we are also interested in different ways of how to efficiently address the problem of propagating bounds and how better collapsing inequalities can be generated.

When improving the use Vampire's AVATAR architecture, besides the integration of state-of-the-art SAT solver we are also interested in investigating whether further refinements and fine tuning of the SAT solver part will produce better overall performance of the theorem prover. We are also interested in investigating different ways of combining the Vampire built-in SAT solver with the external SAT solver such that we do not restart upon every newly added clause. Besides splitting, Vampire uses SAT solving also for instance generation [69] and indexing. Hence, finding out whether the use of a state of

the art SAT solver improves the performance of Vampire in general, constitutes another interesting problem that should be further investigated.

We believe that further extending our approach in order to better integrate theory-specific reasoning engines and better techniques for minimizing the set of invariants will drastically improve the quality of generated invariants. Further improvements can also be achieved by fine tuning the underlying SAT solver. Additionally, interleaving SAT solving steps with steps in the bound propagation method is another interesting topic worth studying.

Bibliography

[1] Clang homepage. http://clang.llvm.org.

[2] Tar homepage. http://www.gnu.org/software/tar/.

[3] http://fmv.jku.at/lingeling/. Lingeling, Plingeling and Treengeling.

[4] http://gmplib.org. The GNU Multiple Precission Arithmetic Library.

[5] http://hpc.uvt.ro/infrastructure/infragrid/. HPC Center - West University of Timisoara.

[6] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. Discount: A system for distributed equational deduction. In *RTA*, pages 397–402, 1995.

[7] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.

[8] Luís Baptista and João P. Marques Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, pages 489–494, 2000.

[9] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

[10] Clark Barrett and Sergey Berezin. Cvc lite: A new implementation of the cooperating validity checker. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer Berlin Heidelberg, 2004.

[11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[12] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, 2008.

[13] A. Belov, D. Diepold, M. M. J. Heule, and M. Järvisalo. Proceedings of sat competition 2014; solver and benchmark descriptions. 2014.

[14] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of PLDI*, 2007.

[15] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 378–394, 2007.

[16] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[17] Armin Biere. Lingeling, plingeling and treengeling entering sat competition 2013. In *SAT Competition 2013*, pages 51–52, 2013.

[18] Armin Biere, Ioan Dragan, Laura Kovács, and Andrei Voronkov. Experimenting with SAT solvers in vampire. In *Human-Inspired Computing and Its Applications - 13th Mexican International Conference on Artificial Intelligence, MICAI 2014, Tuxtla Gutiérrez, Mexico, November 16-22, 2014. Proceedings, Part I*, pages 431–442, 2014.

[19] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 122–131, 2013.

[20] A. R. Bradley and Z. Manna. *The Calculus of Computation - Decision Procedures with Applications to Verification*. Springer, 2007.

[21] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proc. of ISSAC*, pages 54–60, 2007.

[22] Edmund M. Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 294–303, 1987.

[23] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification,*

12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, pages 154–169, 2000.

[24] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual*. CEA LIST, 2010.

[25] Scott Cotton. Natural Domain SMT: A Preliminary Assessment. In *Proc. of FORMATS*, pages 77–91, 2010.

[26] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proc. of POPL*, pages 84–96, 1978.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[28] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

[29] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[30] Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability*, pages 99–130. 2009.

[31] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[33] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.

[34] Leonardo Mendonça de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 475–490, 2008.

[35] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[36] I. Dillig, T. Dillig, and A. Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. In *Proc. of ESOP*, pages 246–266, 2010.

[37] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection orders for cad. In *ISSAC*, pages 111–118, 2004.

[38] I. Dragan and L. Kovács. Lingva: Generating and proving program properties using Symbol Elimination. In *Proc. of PSI*, page to appear, 2014.

[39] Ioan Dragan, Konstantin Korovin, Laura Kovács, and Andrei Voronkov. Bound propagation for arithmetic reasoning in vampire. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, pages 169–176, 2013.

[40] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. of CAV*, pages 81–94, 2006.

[41] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[42] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004.

[43] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 37–47, 2013.

[44] J. Gailly and M. Adler. The Gzip Homepage. `http://www.gzip.org`, 1991.

[45] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 175–188, 2004.

[46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[47] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, 1998.

[48] C.P. Gomes, H. Kautz, Ashis Sabharwal, and Bart Selman. Satisfiability solvers. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 2, pages 89–121. Elsevier, 2007.

[49] D. Gopan, T. W. Reps, and M. Sagiv. A Framework for Numeric Analysis of Array Operations. In *POPL*, pages 338–350, 2005.

[50] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 120–135, 2009.

[51] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, pages 634–640, 2009.

[52] N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.

[53] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[54] K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. In *Proc. of IJCAR*, pages 188–195, 2010.

[55] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in vampire. In *TACAS*, pages 60–64, 2011.

[56] Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In *KI*, pages 435–443, 2009.

[57] Krystof Hoder and Andrei Voronkov. The 481 ways to split a clause and deal with propositional variables. In *CADE*, pages 450–464, 2013.

[58] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 355–370, 2012.

[59] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[60] D. Jovanovic and L. de Moura. Cutting to the Chase Solving Linear Integer Arithmetic. In *Proc. of CADE*, pages 338–353, 2011.

[61] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[62] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[63] K. Korovin. iProver - An Instantiation-based Theorem Prover for First-order Logic (System Description). In *Proc. of IJCAR*, volume 5195 of *LNAI*, pages 292–298, 2009.

[64] K. Korovin, N. Tsiskaridze, and A. Voronkov. Conflict Resolution. In *Proc. of CP*, pages 509–523, 2009.

[65] K. Korovin, N. Tsiskaridze, and A. Voronkov. Implementing Conflict Resolution. In *Ershov Memorial Conference*, volume 7162 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2012.

[66] K. Korovin and A. Voronkov. GoRRiLA and Hard Reality. In *Proc. of Ershov Memorial Conference (PSI)*, pages 243–250, 2011.

[67] K. Korovin and A. Voronkov. Solving Systems of Linear Inequalities by Bound Propagation. In *Proc. of CADE*, pages 369–383, 2011.

[68] K. Korovin and A. Voronkov. Solving Systems of Linear Inequalities by Bound Propagation, full version (2011). 2011.

[69] Konstantin Korovin. Inst-gen - a modular approach to instantiation-based automated reasoning. In *Programming Logics*, pages 239–270, 2013.

[70] L. Kovács and A. Voronkov. Interpolation and Symbol Elimination. In *Proc. of CADE*, pages 199–213, 2009.

[71] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, pages 470–485, 2009.

[72] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *CAV*, pages 1–35, 2013.

[73] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of CGO*, pages 75–88, 2004.

[74] M. Ludwig and U. Waldmann. An Extension of the Knuth-Bendix Ordering with LPO-Like Properties. In N. Dershowitz and A. Voronkov, editors, *LPAR*, Lecture Notes in Computer Science, pages 348–362. Springer, 2007.

[75] Ewing L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, pages 96–106, 1992.

[76] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[77] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[78] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[79] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.

[80] K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to Richer Logics. In *Proc. of CAV*, pages 462–476, 2009.

[81] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535, 2001.

[82] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[83] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 7, pages 371–443. Elsevier Science, Amsterdam, 2001.

[84] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. In *IJCAR*, pages 257–271, 2001.

[85] Valentin Perrelle and Nicolas Halbwachs. An analysis of permutations in arrays. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, pages 279–294, 2010.

[86] Niloofar Razavi, Azadeh Farzan, and Andreas Holzer. Bounded-interference sequentialization for testing concurrent programs. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 372–387, 2012.

[87] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[88] Alexandre Riazanov and Andrei Voronkov. Adaptive saturation-based reasoning. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*, pages 95–108, 2001.

[89] Alexandre Riazanov and Andrei Voronkov. Splitting without backtracking. In *IJCAI*, pages 611–617, 2001.

[90] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.

[91] A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*, volume 1. Elsevier Science, Amsterdam, 2001.

[92] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[93] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and sons, 1998.

[94] Stephan Schulz. System description: E 1.8. In *LPAR*, pages 735–743, 2013.

[95] J. Seward. The Bzip Homepage. `http://www.bzip.org`, 1996.

[96] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theor. Comput. Sci.*, 18:227–258, 1982.

[97] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 131–153. 2009.

[98] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[99] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 223–234, 2009.

[100] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artif. Intell.*, 9(2):135–196, 1977.

[101] Geoff Sutcliffe. Tptp, tstp, casc, etc. In *CSR*, pages 6–22, 2007.

[102] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[103] Andrei Voronkov. Avatar: The architecture for first-order theorem provers. In *CAV*, pages 696–710, 2014.

[104] W.Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. In *Journal of Symbolic Logic 22, no. 3*, pages 269–285, 1957.

[105] Christoph Weidenbach. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning*, pages 1965–2013. 2001.

[106] Hantao Zhang. SATO: an efficient propositional prover. In *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13-17, 1997, Proceedings*, pages 272–275, 1997.

[107] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Curriculum Vitae

IOAN-DUMITRU DRĂGAN

PERSONAL DATA

Date of Birth:	February 11, 1987
Citizenship:	Romanian
Languages spoken:	Romanian (native), English (professional working proficiency, C1), German (medium, B2), French (basic, A1)
Email:	ioan.dumitru.dragan@gmail.com
Webpage:	www.complang.tuwien.ac.at/ioan
Phone:	+43-(0)680-5047660

RESEARCH INTERESTS

Formal software verification, especially assertion synthesis. Automated reasoning, especially automated first-order theorem proving.

EDUCATION

Phd candidate	Vienna University of Technology Austria	Oct 2011 - Present
Master in computer science	West University of Timişoara Romania	Oct 2009 - Jul 2011
Erasmus student	Johannes Kepler University Linz Austria	Oct 2008 - Feb 2009
Bachelor in computer science	West University of Timişoara Romania	Oct 2006 - Jul 2009

EXPERIENCE

Project Assistent — Vienna University of Technology, Vienna, Austria — Oct 2011-Present

My work is funded by the Austrian National Research Network Rigorous Systems Engineering - RiSE project (`arise.or.at`) from the Austrian Science Fund (FWF). The main topic I am currently working on is extending the first-order theorem prover Vampire (`www.vprover.org`) with theory support.

Intern — INRIA Sophia Antipolis, Nice, France — Sept 2010 - June 2011

Duties were related to the implementation of different image processing algorithms using the GPGPU paradigm. The work was focused on image segmentation using probabilistic models.

Intern — IeAT, Timişoara, Romania — Oct 2009 - June 2010

Duties included the implementation of different image processing algorithms. I have successfully implemented different variations of segmentation algorithms to work on distributed systems.

Internship — Bosch, Braga, Portugal — June 2013 - July 2013

Duties included research and documentation regarding parking assistents for vehicles with trailers. Main duties were related to geometrical modeling of the trajectory for this kind of systems.

COMPUTER SKILLS

Extensive knowledge of different operating systems.
Proficient programming skills in C, C++, MPI and scripting languages.
Intermediate proficiency in programming CUDA, Java, C# and openMP.

LIST OF PUBLICATIONS

P1. Ioan Dragan, Konstantin Korovin, Laura Kovács, Andrei Voronkov. Bound Propagation for Arithmetic Reasoning in Vampire. In proceedings of SYNASC 2013

P2. Ioan Dragan, Laura Kovács, Lingva: Generating and Proving Program Properties using Symbol Elimination. To appear in PSI 2014.

A1. Armin Biere, Ioan Dragan, Laura Kovács, Andrei Voronkov, Integrating SAT solvers in Vampire, Vampire Workshop 2014, (Abstract).

P3. Armin Biere, Ioan Dragan, Laura Kovács, Andrei Voronkov, Experimenting with SAT solvers in Vampire, In proceedings of MICAI 2014.

P4. Roland Lezuo, Ioan Dragan, Gerg Barany and Andreas Krall, vanHelsing: A Fast Theorem Prover for Debuggable Compiler Verification, under submission

SCIENTIFIC TALKS

March 2011	Vienna University of Technology	Austria
May 2011	INRIA Sophia Antipolis	France
May 2013	Alpine Verification Meeting, FBK Trento	Italy
Sept. 2013	SYNASC 2013, Timisoara	Romania
May 2014	Alpine Verification Meeting, Frejus	France
June 2014	Chalmers , Gothenburg	Sweden
July 2014	PSI 2014, St. Petersburg	Russia
July 2014	Vampire Workshop , Vienna	Austria
Sept. 2014	RiSE/Puma Meeting, Mondsee	Austria
Nov. 2014	MICAI 2014, Tuxtla Gutierrez	Mexic

SERVICE TO THE SCIENTIFIC COMMUNITY

- Member of the local organizing team of the 9th International Conference on Mathematical Problems in Engineering, Aerospace and Sciences (ICNPAA 2012), Vienna, Austria.
- Member of the local organizing team of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008 and 2009), Timişoara, Romania.
- Volunteer at Vienna Summer of Logic, 2014
- Subreviewer for 11th International Conference on Software Engineering Research, Management and Applications (SERA 2013).
- Subreviewer for 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR19).
- Subreviewer for 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014).
- Subreviewer for Haifa Verification Conference (HVC 2014).
- Subreviewer for Ershov Informatics Conference (PSI 2014).
- Subreviewer for 6th International Symposium on Symbolic Computation in Software Science (SCSS 2014).

EXTRACURRICULAR ACTIVITIES

- Work and travel NH, USA (2007).
- Active member Kolping Banat, Timişoara, Romania (2006-2009).
- Microsoft Student Partner, West University of Timişoara, Romania (2006-2008).