An Evaluation of Symbol Elimination for Generating First-Order Loop Invariants

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of
Diplom-Ingenieurin

in

Computational Intelligence

by

Ioana Jucu
Registration Number 1128547

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Dr. Laura Kovács

Date: 06.10.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Ioana Jucu
Martir Herman Sporer, Timisoara, Timis, Romania

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 Verfasserin)

Acknowledgements

I would like to express my very great appreciation to Dr. Laura Kovács for her valuable and constructive suggestions during the planning and development of this research work. Her willingness to give her time so generously has been very much appreciated.

I would also like to thank Mr. Ioan Drăgan for his support with one of the tools needed.

Abstract

Invariant genereation is a critical problem in proving different properties for programs with loops, properties including correctnes. The problem becomes harder with the incresing numbers of quantifiers in the property to be proven. In this paper we study and combine different methods of invariant generation in order to obtain stronger properties.

Kurzfassung

Invariant generiert ist ein kritische Problem für Programmen mit Schleife zum Beweisen der Eigenschaften, inclusive die Richtigkeit. Die problem wird schwerer bei hohe Anzhal des Quantoren in die geprüfte Eigenschaft. In diese arbeit wir studiere diese Problem und versuchen combinieren verschieden Methoden für schwarer invariants zu beweisen.

Contents

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Propositional logic 3
2.2 SAT solvers 4
2.3 Boogie 10

3 Overview of Invariant Generation Methods **13**
3.1 GinPink 13
3.2 Lingva 15
3.3 CppInv 17

4 Experiments **22**
4.1 Experiments with Gin-Pink and Cpp-Inv 22
4.2 Experiments with Lingva 28
4.3 Discussions of Experimental Results 37

5 Invariant Specific Theory Extensions to First Order Theorem Prover **41**
5.1 Comparison of invariants strength 41
5.2 Discussions and Conclusions 53

6 Conclusions **55**

7 References **57**

1 Introduction

The complexity of software systems is in a continuous grow. Making sure that different pieces of the same system will work together properly is a hard task. The development of software systems imply the work of more people working at different parts of it, using computer, networks, physical devices, and over millions of lines of code in various languages. Integrating, understanding and ensuring the reliability of such a system are necessary task in order to make it useful.

In this paper we give attention to the task of ensuring reliability. In the past years a lot of interest was given to this task and there were developed different methods to do it, but one challenge that was not yet overcome is the analysis of loops. In particular programs dealing with arrays use loops to process the elements, and on the strength to the unbounded nature of this data structures analyzing and inferring properties for elements becomes a challenging problem on its own.

One way to approach this problem is to bound the loop [BCC+03], unfolding it just a limited number of times and afterwards analyzing the new obtained program as if no loop would occur in it. Although this approach is successfully used in model checking techniques, the limitations of applying it consist in the loss of completeness of the algorithm. An informal explanation of this fact is that obtaining a proof that a bounded subset of elements do not have a certain property is a result strong enough to consider that the property does not hold for the entire loop, but if the property holds for the first n unrollings of the loop it may be the case that it will be falsified in one of the following iterations that were cut off by the bound.

Another approach to reason about program loops is to statically analyze the code and extract loop properties automatically. In this thesis we follow this approach. We are going to analyze and compare three methods that automatically extract properties for loops. These methods are the symbol elimination method of [KV09], the constraint-based invariant generation approach of [LRCR13], and the postcondition-based method of [FM10]. We analyze and compare these approaches on series of challenging academic examples which are considered difficult to reason about.

The symbol elimination technique is an automatically mechanism that generates invariants based on the static analysis of programs, that does not require other information from the user about the code analyzed. For this method we use a saturation theorem prover and we choose *Vampire* not only for it's capability to reason with different theories but also due to the fact that is one of the fastest provers awarded several times in competitions.

The constraint-based method first analyses the code discovering every possible path and checking if properties hold in some key points along them. In the next chapters we are going to explain where and why a point in the path becomes such a key point. Properties that are checked are constructed

based on a template following some rules that will also be presented later in the paper.

The third method is the only one from the three presented that needs additional input from the user, namely a postcondition in the form of a formula. The program makes certain changes in the formula, that will be described later on, in order to find valid invariants for the program.

Using the invariants discovered by the constraint-based and postcondition-based methods, in this thesis we further extend the power of symbol elimination in order to reason about more complex loops and invariants than in [KV09]. For doing so, we strengthen the underlining first-order theory reasoning engine of symbol elimination and add additional mathematical theorems and axioms to the symbol elimination problem. The symbol elimination problem is then further fed into a saturation theorem prover and is successfully used to prove the intended loop invariants and properties of the program. Our results show that theory reasoning in first-order theorem proving is a very challenging problem and requires a good understanding of the necessary theory axiomatizations.

2 Preliminaries

In this section we give a brief overview of *SAT solving*, *SMT solving* and *Saturation theorem prover*. Also we give insight of the mechanism of *Boogie* theorem prover. These are necessary for further understanding the mechanism of the tools we are studying in this paper, and since all of them have at their core first-order logic we also present its syntax and semantics.

We present all the above in a step wise manner starting with propositional logic. Based on the syntax and semantic of propositional logic it is easier to understand the ones of first-order logic, since the latter one extends the expressive power of the former by introducing new characters and concepts.

In the same step wise manner we present SAT-solving and SMT-solving. The first problem is related to propositional logic, while the second one makes use of the results obtained in SAT-solving as a subroutine in the algorithm for solving problems encoded in first-order logic.

Also we present the mechanism of a saturation theorem prover, The understanding of this concept is necessary in order to understand the differences between the methods of invariant generation and the properties that make them efficient in different types of problems.

Another point that is touched in this chapter is the higher order logic theorem prover *Boogie*, that has more expressive power than the other tools introduced in this paper and also uses a language specially created for it that has a syntax harder to understand than the others.

We are going to present how these concept interact to each other and with the underlying theory, relating their results with the concept of program verification.

2.1 Propositional logic

Is a part of mathematics important for program verification. The least complex component in the syntax of propositional [DW50] logic is an *atom*. Every atom has a truth value, either true or false, and it represents a statement (such as " Roses are red.") without taking into account its internal structure.

In order to get more complex propositions the atoms can be connected with *connectives*. The propositional syntax is defined as follows:

1. If p is an atom, then p is a proposition;
2. $\top$ and $\bot$ are propositions;
3. If p is a proposition, $\neg p$ is a proposition;
4. if p and q are propositions, then $p \circ q$ is a proposition, where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Given two propositions, p and q, the following statements hold for the described connectives:

- $\neg$ ("negation"), if $\neg p$ is true if p is false, and false otherwise;
- $\wedge$("and"), if p is true and q is true then $p \wedge q$ is true, otherwise $p \wedge q$ is false;
- $\vee$ ("or"), if one of the the two propositions are set to true then $p \vee q$ is true, otherwise $p \vee q$ is false;
- $\Rightarrow$ ("if...then"), if p is true and q is true, or if q is false $p \Rightarrow q$ is true, otherwise is false;
- $\Leftrightarrow$ ("if and only if...then"), if p and q are set to the same truth value then $p \Leftrightarrow q$ is true, otherwise is false;

To make the propositions easier to read and write, and to save parentheses the connectives have priorities as follows: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$(meaning that $\neg$ binds stronger than $\wedge$, $\wedge$ binds stronger than $\vee$, etc.).

The problem of determining if the atoms that form a proposition can be given truth values in such a way that the proposition would evaluate to true is called *satisfiability* problem (abbreviated as SAT).

Example: p, r, q are atoms in propositional logic, and $p \vee q \Rightarrow r \vee q$ is a syntax correct formula. Choosing the values as follows: $p \rightarrow false$, $r \rightarrow false$ and $q \rightarrow true$ will cause the formula to evaluate to $true$ (since $p \vee q$ evaluates to $true$, $r \vee q$ evaluates to $true$).

Satisfiability problem is an NP-complete problem[Coo71]. The complexity and nature of this problem makes it useful in modeling different problems such as digital circuits, constraint satisfaction problems, reasoning about specifications.

2.2 SAT solvers

These are tools that are constructed to solve SAT problems [ES04]. Despite the high complexity of the problem, very good results were obtained in practice with yearly improvements of the solving algorithms and heuristics for the SAT-solving contest.

In figure 1 we present a scheme on how a SAT solver is used to solve a problem.

The first step is to abstract the problem into a set of propositions. Usually the problem can not be abstracted straightforward so some simplification is needed. After finding a suitable representation of the problem as a set of formulae use a SAT solver to find a solution. If a solution is not find specific improvements are made in order to get a more efficient search.

The SAT solvers take the input formulae in a special form named *conjunctive normal form*. Every propositional formula can be transformed in

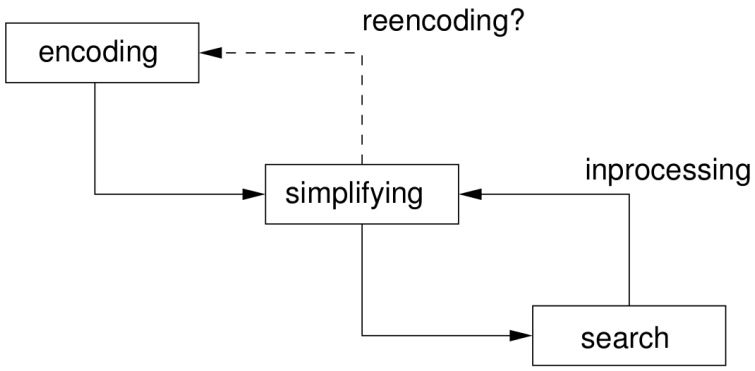

Figure 1: SAT solver structure

an equi-satisfiable formula that satisfies the conditions of the CNF [G.S83]. A formula is said to be in conjunctive normal form if it is a conjunction of *clauses*, where a clause is a disjunction form from atoms or negation of atoms.

The next algorithm called *boolean constant propagation*(BCP) [TH06] is a part of the algorithm that stays at the basis of a lot of modern SAT solvers.This is a very short algorithm with three steps that repeat as long as possible:

1. find clause that contains a single literal (also named *unit clause*);
2. eliminate all clauses that contain the literal;
3. in all other clauses eliminate the occurrence of the negation of the literal;

The DPLL[DL62] algorithm was developed in 1962 and still used to this day in the state of the art SAT solvers. One variant of the algorithm is this:

```
DPLL(F)
  F := BCP(F)
  if F =⊤
      return satisfiable
  if ⊥∈ F
      return unsatisfiable
  pick remaining variable x and literal l ∈ {x, ¬ x}
  if DPLL(F ∧ {l}) returns satisfiable
      return satisfiable
return DPLL(F ∧ {¬l})
```

The idea of the algorithm is simple and efficient. First boolean constant propagation is applied, if there are no more clauses remaining, the algorithm

returns satisfiable, if a clause becomes empty after this step the algorithm returns unsatisfiable. Otherwise a literal is selected from the remaining clauses and DPLL is called on these clauses plus an extra unit clause containing the selected literal.

The selection of the literal has a great significance in the efficiency of the SMT solver. A lot of heuristics were developed an studied in order to get better results. These are a few ideas that were used for the heuristics:

- Dynamic Largest Individual Sum[MSS99]: the literal that appears most often in the unsatisfied clauses at the current point is chosen;
- Jeroslov-Wang heuristic[Wan95]: the choices are made such that a unit clause is soon obtained;
- VSIDS[MMZ$^+$11]: is a complex heuristic that takes into account partial assignments that are unsatisfiable, and based on a directed acyclic graph of the solution it chooses a literal that is part of the unsatisfied clause;

Although SAT cover a significant range of problems, propositional logic is not expressive enough to cover other problems of practical interest, this is why *first order logic* got attention for this purpose. Since the problem of satisfiability is harder in this logic there were restrictions made with respect to some *theory*. This new problem is referred to as *Satisfiability Modulo Theory (SMT)*[dMB11].

The first order logic (FOL) syntax is more complex[And02]. There are two extra logical symbols to propositional logic: the quantifiers $\forall$ (universal- for representing judgments that are true for all objects) and $\exists$ (existential- for representing particular judgments). There are also two other types of symbols: *functions* and *predicates*. Function symbols together with the predicates symbols along with their arity form the *signature*.

Terms in FOL are defined inductively as follows:

- every variable is a term;
- if $t_1, t_2 ... t_n$ are terms and f is a function symbol with arity n then $f(t_1, t_2, ... t_n)$ is a term;

The *signature* is a countably set of predicate symbols and function symbols together with their arity.

The set of formulas are defined inductively in the following way:

- $\top$ and $\bot$ are formulae;
- if $t_1, t_2 ... t_n$ are terms and p is a function symbol with arity n then $p(t_1, t_2, ... t_n)$ is a formula;
- if ϕ is a formula then $\neg\phi$ is a formula;

- if ϕ and φ are formulae then also $\phi \circ \varphi$ is a formula where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$;
- if ϕ is a formula and x a variable then $\exists x\phi$ and $\forall x\phi$ are formulae;

A variable is called *free* if it is not bounded by a quantifier ($\exists$, $\forall$).

Example: $\forall x, p(x) \Rightarrow p(y)$, x is bounded by the existential quantifier while y is a free variable.

The semantics of a first order logic formula is given by an *interpretation*. The interpretation consist of an non-empty domain U and an interpretation function $I()$.

The interpretation function maps in the following way given the domain U:

- every function symbol of arity 0 (*constant symbol*) with a element from the domain;
- every for every function symbol f with arity > 0 $I(f) : U^n \rightarrow U$ (for every n combination of terms the value of the function for the combination is a value in the domain);
- for every predicate symbol p with arity n $I(p) : U^n \rightarrow \{1, 0\}$ (for every n combination of terms the value of the function for the combination is a value in the set $\{1, 0\}$, 1 representing $true$ and 0 representing $false$)

The free variables can be interpreted in two ways: either universally quantify the formula or bound the variable to a constant in the domain.

A theory in first order logic is considered any set of formulae that do not contain free variables[DP60]. A formula without any free variables is also called *sentence*.

A SMT-solver is a tool that takes as input a set of first order formulae and gives as output the answer satisfiable or unsatisfiable, taking into account information and methods of some first order theories when needed. In figure 2 there is the structure of a modern SMT-solver [DdM06]. The *core solver* in the figure refers to a solver that can solve the satisfiability problem for equality and uninterpreted functions theory, while the *satellite solver* handles other theories such as arithmetical, arrays, bit vectors, or data types. Let there be observed that the satellite solver exchanges information only with the core solver, the core solver communicates both with the satellite solver and the SAT-solver, and the SAT solver communicates only with the core solver.

A general algorithm for SMT solver [AAR09]is presented in the following lines:

SAT−value SMT_solver (T−formula ϕ){
$\phi' = convert_to_cnf(\phi)$

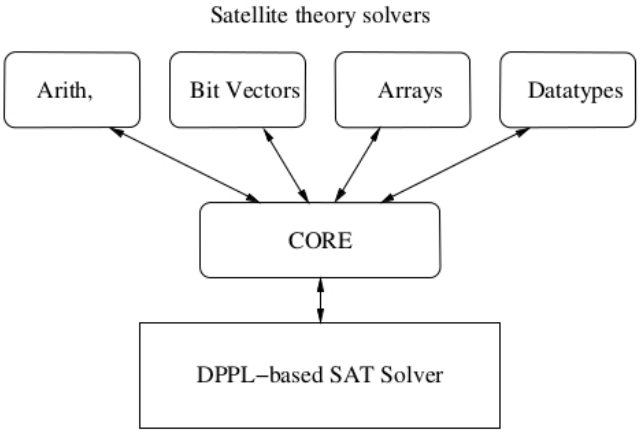

Figure 2: SMT solver structure

```
φ^p = T2P(φ')
while (DPLL(φ^p, μ^p) == SAT) {
⟨ρ, η⟩ = T − solver(P2T(μ^p))
    if (ρ == SAT) return sat
    φ^p = φ^p ∧ T2P(¬η)
    }
    return unsat
}
```

The algorithm takes as input a a formula in the theory T an outputs satisfiable or unsatisfiable. The first step is to convert the formula in it's CNF form and store this form in a new variable $\phi^{'}$. $\phi^{'}$ is abstracted into its propositional form (by the $(T2P)$ function) and stores the new propositional formula in ϕ^p. The DPLL algorithm takes as input the propositional formula and either returns unsatisfiable which makes the initial algorithm to return unsatisfiable, or it returns a satisfying assignment on the literals of the formula. The $T-$solver then checks the mapping of the formula back to the theory (with the assignment proposed by SAT-solver) and either returns satisfiable, which causes the main algorithm to exit with the status satisfiable, or it returns a set η of literals that caused an inconsistency in the theory. η is abstractive to propositional logic and its negation is conjuncted with the rest of the abstractisation of the formula formula, and DPLL is called again on the new obtained formula.

Given a hypotheses (in this case this is the formula) if we can reach the empty set (refutation) by using an *inference system*, this would give us a *refutation proof*. An inference system is a set of *inference rules*. An inference rule is can be described as being an n-ary relation on formulas, with $n \geq 0$. The elements of such relations are called inferences and usually written as $\frac{F_1 \ldots F_n}{F}$.

Algorithms were developed in order to obtain interpolants and invariants from the proofs of unsatisfiability outputted by an SMT-solver.

A theorem prover is a tool used to prove theorems in different logics. A specific type of theorem provers are the saturation theorem provers. We say that a set of formulas is saturated with respect to an inference system I if we can find another set of formulas containing the initial one that is closed under inference with respect to I[KV09].

The saturation theorem prover. *Vampire*, used for this study is using a *superposition* inference system. In order to give a brief description of this system first we introduce the notion of *simplification ordering* on terms[KVar]. If an ordering $\succ$ has the following properties it is considered a simplification ordering:

- is well-founded, that is there exists no infinite sequence of terms $t_0, t_1, \ldots$ such that $t_0 \succ t_1 \succ \ldots$;
- is monotonic: if $l \succ r$, then $s[l] \succ s[r]$ for all terms s, l, r;
- is stable under substitutions: if $l \succ r$, then $l\theta \succ r\theta$ (where a substitution θ is considered a simultaneously replacement of all occurrences of a set of terms with another corresponding set of terms,respectively, in a formula);
- has the subterm property: if r is a subterm of l and $l \neq r$, then $l \succ r$.

A *selection function* is a function that selects one or more literals from a non-empty clause. In what follows, selected literals will be underlined (if L is a selected literal then it would be written as $\underline{L}$). *A unifier* of two expression is a substitution that would make the expressions equal. The inference rules for a superposition inference system are the following:

Resolution:

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

where θ is a mgu of A and A'.

Factoring:

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

where θ is a mgu of A and A'.

Superposition:

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

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

where the following hold: θ is an mgu for l and s, s is not a variable, $r\theta \not\succ l\theta$, in the first rule $L[s]$ is not an quality literal, in the last two rules $t'\theta \not\succ t[s]\theta$.

Equality Resolution:

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

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 an mgu of s and s', $t\theta \not\succ s\theta$, $t'\theta \not\succ t\theta$.

A set S of clauses is saturated with respect to an inference system if for every possible combination of the clauses and for every rule in the system, a clause that is already in the system is inferred. A saturation algorithm is considered fair if all possible combinations of clauses and every rule get a chance to be applied at one point. In order for such an algorithm to be useful in practice it needs to be*sound* and *complete*. A complete saturation algorithm will eventually derive the empty clause if the set of clauses is unsatisfiable, and a sound saturation algorithm will correctly conclude that the set of clauses is unsatisfiable if the empty clause is derivable from it. A complete and sound saturation algorithm can have the following outputs in practice:

- unsatisfiable, if the empty clause is generated;
- satisfiable, if the set of clauses is saturated;
- unknown, if the algorithm runs forever (until it runs out of resources) and the empty clause is not derived.

2.3 Boogie

Is a modular reusable verifier for object-oriented Programs [BCD$^+$05]. This tool is made from different components: a source programming language, its usage rules and formal semantics, a logical encoding suitable for automatic reasoning, abstract domains for program analysis and property inference, decision procedures for discharging proof obligations, and a user interface that lets a user understand the results of the verification process[BMSW10].

A representation of how all the components of Boogie interact is represented in figure 3.

The *source programming language*, Spec#, is a high-level, strong typed language. It is a superset of the $C\#$ programming language, giving also

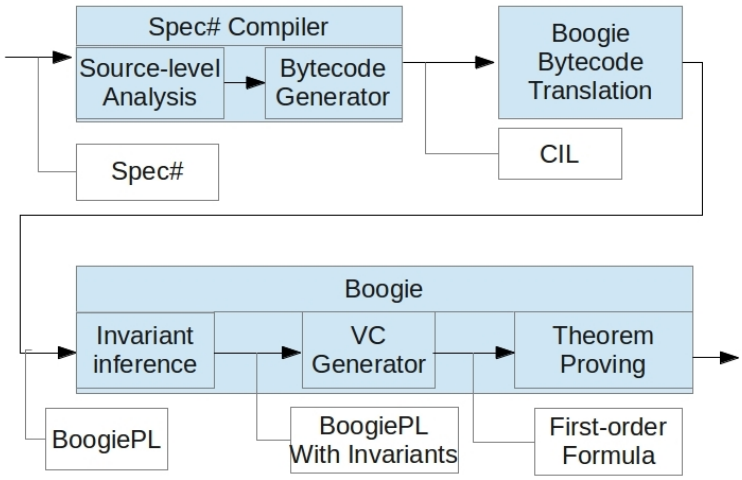

Figure 3: Boogie pipeline

the possibility to write preconditions, postconditions and object invariants. Apart from the compilers checks for static types, it also creates conditions for the dynamic types as part of the target code. Boogie tries to checks statically also the dynamic types properties enforced by the compiler along with the ones provided by the user and those defined by the virtual machine. The source code is compiled into CIL language.

The CIL code is obtained from an abstract syntax tree either directly from the compiler, which enables Boogie to work as part of the compiler and offer information to the user in a design-time manner, or from an already compiled .dll or .exe file.

The *intermediate language* is obtained by translating the CIL code in BoogiePL code. This process enables the writing of new statements: *assert* and *assume*. The assert statements are encode conditions that will be checked by the program verifier, and the assume statements encode properties that can be used by the verifier, these properties being enforced by the source language and the verification process. Also BoogiePL permits the encoding of theories and mathematical symbols. Since BoogiePL has a textual representation small changes can be made in this file without damaging the Spec# code. Also the textual representation makes Boogie useful for other verifier, making the verification conditions reusable. At this point the code is replaced with the proof task.

The BoogiePL code is transformed into first order logic properties. For this process loop invariants are needed, and since providing this by hand is troublesome and sometimes impossible, Boogie offers a framework that automatically infers loop invariants from BoogiePL code, written in the form of "assume" statements.

The next step is to get verification condition from every basic block of

the program. These are written in first order logic with arithmetic, and since there are more ways to write the same condition, the chosen form affects the performance of the theorem prover. Also the encoding of the conditions are made in such a way that if the verification fails a trace of failure can be mapped back in the original input language. A failure in verification can also be only spurious since the theorem prover is incomplete, and also it might be the case the the theorem prover could not do the task due to the fact that there were not enough resources.

3 Overview of Invariant Generation Methods

There is a large variety of invariant generation approaches researched in the past years. In this chapter we present three of this methods which cover a large area of invariants that can be inferred and represent the state of the art in their representative domain. The tools implementing this methods were made available by their respective researchers. Although other methods were considered for this thesis the tools implementing them were not available at the moment from different reasons. The approach suggested has one of the following starting points for obtaining the result: post-conditions, saturation theorem proving, use of predefined templates. In what follows we are going to give an insight of the idea used in each of the three methods.

3.1 GinPink

This method makes use of the postconditions provided by the user in order to find an invariant for a certain loop in the procedure [FM10].

There are four different heuristics that are used to weaken the postcondition: constant relaxation, variable aging, uncoupling, term dropping.

Constant relaxation replaces a constant in the postcondition with a variable. A constant is considered a variable that is not modified by the loop, and a variable - in this context- is a variable modified by the loop. An example were this heuristic is used is:

```
procedure ArrayInit <tt> ( A: array tt, left: int,
               right: int, index: int) returns (i:int)
requires left <= right;
ensures (forall k: int :: k != n  ==>  A[k] == 0);

{ var i: int;
i:= left; index:= left; A[left]=0;
while(i<= right)
invariant (index <= i);
{
i=i+1;
A[i]:=0;
}
}
```

In this program all elements of an array are initialized with the value 0. By relaxing the constant n in the postcondition by the variable i an invariant is obtained.

In some cases just substituting a constant with a variable does not yield an invariant, depending on the update time of the variable. *Variable aging* replaces a constant with an expression involving a variable.

```
procedure ArrayInit <tt> ( A: array tt, left: int,
               right: int, index: int) returns (i:int)
requires left <= right;
ensures (forall k: int :: k != n  ==>  A[k] == 0);

{ var i: int;
i:= left; index:= left;
while(i<= right)
invariant (index <= i);
{
A[i]:=0;
i:= i+1;
}
}
```

Although the above example is similar to the one given for constant relaxation heuristic we observe that the same invariant no longer holds, because of the updating manner of the

When invariants are conjunctions of formulas there might be the case that substituting a constant with one variable on both sides of the conjunction does not result in an invariant. *Uncoupling* is the heuristic that replaces one constant with different variables at different occurrences in the postcondition resulting in an invariant.

```
 partition (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER
    require A.length = n ≤ 1
    local low index, high index : INTEGER
    do
      from low index := 1; high index := n
      until low index = high index
      loop
from | no loop initialization
  until low index = high index ∨ A[low index] > pivot
loop low index := low index + 1 end
from | no loop initialization
until low index = high index ∨ pivot > A[high index]
loop high index := high index  1 end
    A.swap (A, low index, high index)
      end
      if pivot ≤ A[low index] then
  low index := low index  1
  high index := low index
      end
      Result := low index
```

$$\texttt{ensure } (\forall\ k\ \ 1 \leq k \wedge k < Result + 1 = A[k] \leq pivot\ ) \wedge\ (\forall\ k\ \ Result < k \wedge k \leq n = A[k] \geq pivot\ )$$

The postcondition is assumed to be in conjunction normal form so applying *term dropping*, means removing some terms from the formula in order to weaken it. An example where this method is applied is if in the function *Partition* we would like to ensure only $\forall k Result < k \wedge k \leq n = A[k] \geq pivot$. This is still an invariant for the code.

In order to find the invariants for the loop, the algorithm considers first the postcondition without any weakening, and afterwards apply the four mentioned heuristics and check if the resulting formulas are invariants for the loop. The first heuristic applied is constant relaxation. There are to sets of candidates resulting from this step. The first set of candidates is obtained by replacing every occurrence of a constant with the same variable. The second set of candidates are obtained by uncoupled replacement of constants with variables.

This method relies on the Boogie verification tool to generate and prove verification conditions of programs. Proving verification conditions is done by using SMT reasoning.

3.2 Lingva

The idea of this method is to generate invariants of the loop in the form of first order logic formulae, by statically analyzing the code [CC77], and afterwards use theorem prover Vampire to eliminate the auxiliary symbols that occurred.

For this method the *guarded assignments* were introduce[Dij75, MP92]. These are expressions of the form $G \rightarrow \alpha_1; \alpha_2; ...\alpha_m$, where α_i is either a scalar variable assignment or an array variable assignment, and G a formula (called guard).

The guards must be mutually exclusive but at least one of them true in every state. There must not be two assignments guarded by the same expression that can modify the same array term at one point, and the left-hand side of all assignments should be different.

After the loop is transformed such that it consist only of guarded assignments a static analysis tool is used (*Aligator*[HHKR10]) to get invariants over scalar variables. From the invariants obtained loop properties are extracted.

In order to deal with loop invariants there are introduced two predicates, named update predicate: $upd_V(i, p)$ (at iteration i the array V is updated at position p), $upd_V(i, p, x)$ (at iteration i the array v is updated at position p with the value x). These two predicates help us express two key properties for the arrays:

- an element in the array V,$V[p]$, is a constant if V is never updated at position p;
- if an element $V[p]$ of an array V is last updated at iteration i, then this is the iteration in which $V[p]$ gets its final value.

Another property that can be extracted from the loop is *constant array*, meaning that there is an array that is never updated.In this situation, adding in $Vampire$ the property $(\forall i)(A^{(i)} = A^0)$ helps on getting more useful invariants.

Monotonicity properties can also be discovered at scalar variables by using a program analysis tool or a light-weight analysis. Variables can be strictly increasing/decreasing $(\forall i, (v^{(i+1)} > v^{(i)})/\forall i, (v^{(i+1)} < v^{(i)}))$, increasing/decreasing, dense increasing/decreasing (meaning that the value of the variable is changed with at most 1 from the previous iteration).

Another class of properties that can be found are *update properties of monotonic variables*. These properties refer to the fact that there is an iteration i in the loop that modifies a monotonic variable when the value of this variables can be bounded in an interval.

Also the guarded assignments are transformed in properties that can be added to $Vampire$ as theorems in order to get useful invariants.

After getting as many properties as possible $Vampire$ is run on them so it would derive invariants without using the auxiliary functions and predicate symbols that were introduced in order to get scalar variables properties.

$Vampire$ was chosen for this method because it can reason with linear integer arithmetic and it has implemented procedures to eliminate symbols. New axioms were introduced in $Vampire$ in order to be able to deal with integer arithmetic:

- $x \geq y \Leftrightarrow x > y \vee x = y$;
- $x > y \Rightarrow x \neq y$;
- $x \geq y \wedge y \geq z \Rightarrow x \geq z$;
- $s(x) > x$;(where s(x) is the successor function);
- $x \geq s(y) \Rightarrow x > y$;(where s(x) is the successor function).

These new axioms enable a sound but incomplete reasoning.

Another trick is used to make $Vampire$ deal with symbol elimination. First there are introduced new axioms for every assignment that has on the left hand-side a variable v: $v^{(0)} = v_0$, $v^{(n)} = v'$, where $v^{(0)}, v^{(n)}$ are representing variable v before the first iteration and after n-th iteration respectively. The newly introduced symbols are called *target symbols*. The goal is to derive only classes that contain only target symbols, interpreted symbols or skolem functions, but at least one target symbol or skolem function. Giving

high precedence in the algorithm used by *Vampire* too the symbols that are not interesting in deriving the new clauses, vampire will eliminate these first.

Every clause derived that respects the above conditions and do not contain a skolem function is an invariant.

TPTP is the language used by *Vampire* so in order to encode problems and to understand the proofs outputted we give a short table with the correspondence between the first-order logic symbols and mathematical symbols and TPTP:

first-order logic and mathematics	TPTP
$\top$	$\$true$
$\bot$	$\$false$
$F_1 \wedge \ldots \wedge F_n$	$F_1 \& \ldots \& F_n$
$F_1 \vee \ldots \vee F_n$	$F_1 \mid \ldots \mid F_n$
$F_1 \rightarrow F_2$	$F_1 => F_2$
$F_1 \leftrightarrow F_1$	$F_1 <=> F_2$
$\forall x_1 \ldots \forall x_n F$	$![X_1 \ldots X_n] : F$
$\exists x_1 \ldots \exists x_n F$	$?[X_1 \ldots X_n] : F$
$-a$	$\$uminus(a)$
$a \leq b$	$\$lesseq(a, b)$
$a < b$	$a < b$
$a + b$	$\$sum(a, b)$
$a * b$	$\$product(a, b)$

3.3 CppInv

The basic idea of this method is to use a predefined set of template properties and check which ones are invariants.

The programs are seen as transition systems of the form $\mathcal{P} = \langle \overline{u}, \mathcal{L}, l_0, \mathcal{T} \rangle$, where $\overline{u}$ is a set of variables, $\mathcal{L}$ is a set of locations, l_0 is the initial location and $\mathcal{T}$ is the set of transitions. A transition is a tuple $\langle l_i, l_j, p\rho_t \rangle$, where l_i, l_j are locations, and ρ_t is a boolean formula representing the transformation of the program variables after the transition.

Take for example the following function that initializes an array with the value 0:

```
 int main()
{
  const int N;
  assume(N >= 0);
  int A[N];

  int i=0;
```

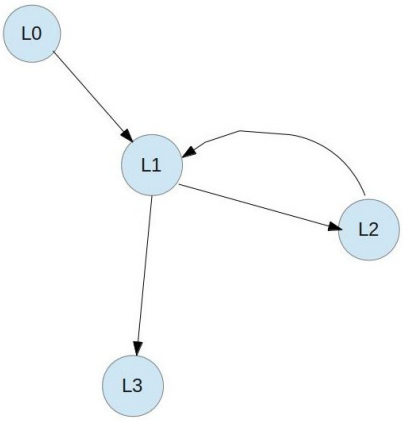

Figure 4: Transition System

```
    while(i < N)
    {
      A[i] = 2*i+3;
      i++;
    }
}
```

The corresponding transition system is represented in figure 4 .

A cyclic path is a path that contains a cycle. A cut-set is a set of locations such that every cyclic path in the graph contains a location that there is also in the cut-set. The locations in the cut-set are called cutpoints.

For this method there are considered the initiation paths, which are the paths in the control-flow graph that connects a location from outside a strongly connected component the a location inside it, and the consecution paths, that label edges only inside the strongly connected component.

Theorem 3.1 *Let $l_1^C, ..., l_p^C$ be a cut-set of a strongly connected components. Let $P_1, ...P_p$ be properties over the program variables $\overline{u}$ such that:*

- *for all initiation paths π^I from l to l_i^C: $\forall \overline{u}, \overline{u'} \rho_{\phi^I} \Rightarrow P_i'$*
- *for all consecution paths π^C from l_j^C to l_i^C: $\forall \overline{u}, \overline{u'} \rho_{\phi^C} \wedge P_j \Rightarrow P_i'$*

Then $P_1, ..., P_p$ are invariants at $l_I, ..., l_p^C$. We say $P_I, ..., P_p$ are inductive invariants.

The semantics of the above theorem is given as follows: We have a set of strongly cut-set for a given strongly connected component and a set of properties over some variables. In order for a property to be considered an invariant there are two conditions that must be fulfilled: for all initial paths to one of the cutpoints, for all variables the formula describing the transition

relation between the initial value of the variable and the value obtained after the transition must imply the corresponding property. The second condition is that for consecution paths from one point l_i^C to l_j^C, for all variables the formula describing the transition relation between the values of the variables from one location to the other in conjunction with the property that holds at the second location (with respect to the value of the variables at the first location), implies the property that holds at the firs location with the values of the variables obtained after the transition.

Using the previous theorem and formalizing the implications as constraints containing both program variables and (not yet known) parameters, invariants can be obtained by finding a solution to the constraints.

For arrays the method generates invariants of the form:
$\forall \alpha : 0 \leq \alpha \leq C(\overline{v}) - 1 : \sum_{i=1}^{m} \sum_{j=1}^{k} \alpha_{ij} A_i[d_{ij}\alpha + \varepsilon_{ij}(\overline{v})] + \mathcal{B}(\overline{v}) + b_\alpha \alpha \leq 0$
where:

- $C, \varepsilon_{ij} \mathcal{B}$ are polynomials with integer coefficients and variables in $\overline{v} = (v_1, v_2, ..., v_n)$
- $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}, \forall i \in \{1, ..., m\}, j \in \{1, ..., k\}$
- $\overline{a} = (A_1, ..., A_m)$ the tuple of array variables

For computational reasons the invariant generation has three steps:

- expressions C are generated in such a way that the domain $\{0...C-1\}$ is empty after initial paths, and C is increased with at most one after consecution paths;
- find expressions $d_i\alpha + \varepsilon_i$ for every array and every C found such that these are valid access points in the array, after executing a consecution path the already analyzed positions are not changed, and after consecution paths ε_i has the same value or the value $\varepsilon_i - d_i$;
- for every array choose k ε_{ij} that either all stay the same or get new values after consecution paths;
- find $\alpha_{ij}, b_\alpha, \mathcal{B}$ to fulfill the property depending on the which case is ε_{ij};

While theorem 3.1 can be applied in the case of linear scalar properties, this is not the case for the array properties, that is another theorem was formulated for this case:

Theorem 3.2 *Let $C, \mathcal{B}, \varepsilon_{ij}$ be linear polynomials with integer variables over the scalar variables, and $a_{ij}, d_{ij}, b_\alpha \in \mathbb{Z}$ for $i \in \{1, ..., m\}$ and $j \in \{1, ...k\}$. If*

1. *Every initiation path π_r^I with transition relation $\rho_{\phi_r^I}$ satisfies $\rho_{\phi_r^I} \Rightarrow C' = 0$*

2. *For all consecution paths* π_s^C *with transition relation* $\rho_{\phi_s^C}$ *satisfies* $\rho_{\phi_s^C} \Rightarrow C' = C \vee C' = C + 1$

3. *For all consecution paths* π_s^C*, all* $i \in \{1...m\}, j \in \{1...k\}$*,* $\rho_{\phi_s^C} \wedge C' > 0 \Rightarrow 0 \leq \varepsilon'_{ij} \leq |A_i| - 1 \wedge 0 \leq d_{ij}(C' - 1) + \varepsilon'_{ij} \leq |A_i| - 1$

4. *For all consecution paths* π_s^C *either:*

   (a) $\rho_{\phi_s^C} \wedge C' > 0 \Rightarrow \varepsilon'_{ij} = \varepsilon_{ij}$ *for all* $i \in \{1...m\}, j \in \{1...k\}$

   (b) $\rho_{\phi_s^C} \Rightarrow C' = C + 1 \wedge \varepsilon'_{ij} = \varepsilon_{ij} - d_{ij}$ *for all* $i \in \{1...m\}, j \in \{1...k\}$

5. *For all consecution paths* π_s^C*,* $\rho_{\pi_s^C} \Rightarrow \forall \alpha : 0 \leq \alpha \leq C - 1 : A'[d_{ij}\alpha + \varepsilon_{ij}] = A_i[d_{ij}\alpha + \varepsilon_{ij}]$

6. *For all consecution paths* π_s^C

   - $\rho_{\pi_s^C} \wedge C' = C + 1 \Rightarrow \sum_{i=1}^{m} \sum_{j=1}^{k} \alpha_{ij} A'_i[d_{ij}C + \varepsilon'_{ij}] + \mathcal{B} + b_\alpha C \leq 0)$ *(case 4a)*
   - $\rho_{\pi_s^C} \Rightarrow \sum_{i=1}^{m} \sum_{j=1}^{k} \alpha_{ij} A'_i[\varepsilon'_{ij}] + \mathcal{B}' \leq 0)$ *(case 4b)*

7. *For all consecution paths* π_s^C

   - $\rho_{\pi_s^C} \wedge 0 \leq \alpha \leq C - 1 \wedge x + \mathcal{B} + b_\alpha \leq 0 \Rightarrow x + \mathcal{B}' b_\alpha \alpha \leq 0$ *x is universally quantified fresh variable, when 4a applies*
   - $\rho_{\pi_s^C} \wedge 0 \leq \alpha \leq C - 1 \wedge x + \mathcal{B} + b_\alpha \leq 0 \Rightarrow x + \mathcal{B}' b_\alpha(\alpha + 1) \leq 0$ *x is universally quantified fresh variable, when 4b applies*

then $\forall \alpha : 0 \leq \alpha \leq C - 1 : \sum_{i=1}^{m} \sum_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \varepsilon_{ij}] + \mathcal{B} + b_\alpha \alpha \leq 0$ *is an invariant.*

The theorem above shows the relation between three linear polynomials $\mathcal{C}$, $\mathcal{B}$, $\varepsilon_{i,j}$, with respect to the paths obtained in the transition systems.

The formula expressed by a transition relation from an initial path must imply that polynomial $\mathcal{C}$ with the value of the variables obtained after the transition equals 0.

The rest of the conditions talk only about consecution paths. The polynomial $\mathcal{C}$ can have either the same value or the value increased by one when evaluated with the new values of the variables and the old values.

There are also integer variables a_{ij}, d_{ij}, b_α that are kept under constraints following some rules.

For all consecution paths if the formula expressing the transition relation is true and the polynomial $\mathcal{C}$ with the new value of variables (denoted by $\mathcal{C}'$) is greater than 0 then the value of ε_{ij} (with the new value of variables denoted by ε'_{ij}) is bounded by 0 to the left and by $|A_i| - 1$ to the right, and the sum $d_{ij}(\mathcal{C}' - 1) + \varepsilon'_{ij}$ is bounded by 0 and $|A_i| - 1$.

The fifth condition in the theorem expresses the fact that if the formula expressing the transition relation is true on such a path than also the element of the array at position $d_{ij}\alpha + \varepsilon_{ij}$ has the same value at the start location and at end location.

There are three more conditions in the theorem (4,6,7) which split the cases of the consecution paths in 2 categories. Each of the three conditions express constraints for the polynomials and the variables in the theorem. The final formula combines all of the above expressing an invariant. For an extended proof of this theorem please refer to [LRCR13].

The conditions of the theorem are encoded into an SMT problem, and a SMT solver gives the formula(s) representing the invariants.

4 Experiments

In this section we present the results obtained by running the three tools from Section 3 on a set of loops that were selected from different papers and benchmarks. The experiments reported in this thesis were obatined using machine with an AMD dual-core processor with 800 Mhz, cache of size 256MB. The limitations of this machine did not allow us to reach the full computational power of the tools we tested for some of the loops in question, giving an inconclusive answer. We treated this cases as negative results, and in the case of Lingva we used them in the next section where we try to improve them. Nevertheless these facilities were in most cases enough in order to get a result showing the capabilities of the tools.

Lingva is independent on the platform on which is run, but we used Linux, and the same platform was used, as requested, for $Cpp - inv$. $Gin - pink$ is bounded to Windows platform.

4.1 Experiments with Gin-Pink and Cpp-Inv

The results for this part are presented in a table in the following form: on the first column of the table there are the loops of interest together with the reference to the paper where it can be found. The programs are written in the C language, and the input for the different differ according to the input language required by each of them. We chose to present them in this manner for the clean look and readability.

On the second column of the table on every row there is one or more invariants that were found by the Cpp-inv tool, ran on the corresponding loop. The properties are written in the form of first order formula with mathematics and their characteristics and explanations can be found at the end of this subsection.

The third column of the table corresponds to the invariants found by the tool Gin-Pink on the set of loops. These are also written in the form of first-order formulas and mathematics and occasionally there might be expressions written in the Boogie language. The results are explained later at the end of this subsection.

program	Cpp-inv	Gin-Pink

Initialization [SS09]
a=0;
while(a<m)do
aa[a]=0; a=a+1;
end do
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$aa[a] = 0$
		$\forall k, k \neq i \Rightarrow$
		$aa[k] = 0$

Insertion [LRCR13]
x=aa[i];
j = i-1;
while (j >= 0 and
aa[j] > x) do
aa[j+1] = aa[j];
--j;
end do
	$\forall a, a \leq 0 \leq i-j-2 \Rightarrow$
	$-aa[-a+i]+x+1 \leq 0$
		$\forall k, k \neq i-j-2 \Rightarrow$
		$aa[i-k] \geq x+1$

Partition [SS09]
a=0;
b=0;
c=0;
while (a<m) do
if(aa[a]>=0)
then bb[b]=aa[a];
b=b+1;
else cc[c]=aa[a];
c=c+1;
end if;
a=a+1;
end do
	$\forall x, 0 \leq x \leq c-1 \Rightarrow$
	$cc[x]+1 \leq 0$
	$\forall x, 0 \leq x \leq b-1 \Rightarrow$
	$-bb[x] \leq 0$
		$\forall k, bb[k] == aa[k]$

Maximum [FM10]
int i=1;
int max = aa[0];
while (i<N)
{
if (max<aa[i]) {
max = aa[i];
}
++i;
}
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$aa[a] - max \leq 0$
		$is\_max(m, A, 1, n)$

HeapProperty [LRCR13]
`const int m;`
`//assume(m>=0);`
`int aa[2*m];`
`int i=0;`
`while (2*i+2<2*m)`
`{`
`if (aa[i]>aa[2*i+1] or`
`aa[i]>aa[2*i+2])`
`break;`
`++i;`
`}`
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$-aa[2*a+1]+aa[a] \leq 0;$
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$-aa[2*a+2]+aa[a] \leq 0;$
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$2a-2*aa[2a+1]+2*aa[a]$
	$-2m+3 \leq 0;$
	$-aa[2a+1]+aa[a] \leq 0;$
	$-aa[2*a+2]+aa[a] \leq 0;$
		$\forall a, a \neq i \Rightarrow$
		$aa[a] \leq aa[2a+1]$
		$aa[a] \leq aa[ik+1]$
		$aa[a] \leq aa[2k+i]$

FirstOccurence [LRCR13]
`int aa[m];`
`int x = readX();`
`int l = 0;`
`int u = m;`
 	
`//aa is sorted asscendent`
`while (l < u) {`
`int m = (l+u)/2;`
`if (aa[m] < x) l = m+1;`
`else u = m;`
`}`
	$\forall a, 0 \leq a \leq m-u-1 \Rightarrow$
	$x-aa[m-a-1] \leq 0;$
	$4u+x-aa[m-a-1]-$
	$-2m-2l \leq 0;$
	$\forall a, 0 \leq a \leq l-1 \Rightarrow$
	$aa[a]-x+1 \leq 0;$
	$aa[a]-4l+2u-x+3 \leq 0;$
		$\forall a, a \neq m \Rightarrow$
		$aa[a] \neq x$
		$\forall a, a \neq l \Rightarrow$
		$aa[a] \neq x$
		$\forall a, a \neq m \Rightarrow$
		$aa[a] \neq l$
		$\forall a, a \neq u \Rightarrow$
		$aa[a] \neq x$
		$\forall a, a \neq m \Rightarrow$
		$aa[a] \neq u$

Palindrome [LRCR13]
`int aa[m];`
`int i=0;`
`while (i<m/2)`
`{`
`if (aa[i] != aa[m-i-1])`
`{`
`break;`
`}`
`i++;`
`}`
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$aa[m-a]-aa[a] \leq 0$
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$-aa[m-a]+aa[a] \leq 0$
		$\forall a, a \neq i \Rightarrow$
		$aa[a] == aa[m-a-1]$
		$aa[a] == aa[i-a-1]$
		$aa[a] == aa[m-a-i]$

PartitionInit [SS09]
`int aa[m], bb[m], cc[m];`
`int i=0, c=0;`
`while (i<m)`
`{`
`if (aa[i] == bb[i])`
`{`
`cc[c] = i;`
`c++;`
`}`
`i++;`
`}`
`}`
	$\forall l, 0 \leq l \leq c-1 \Rightarrow$
	$cc[l]-i+c-l \leq 0$
	$l-cc[l] \leq 0$
		$\forall l, l \neq i \Rightarrow$
		$cc[l] \leq l+i-c$

Vararg [SS09]
`a=0;`
`aa[m];`
`while(aa[a]>0 && a<m){`
`a=a+1;`
`}`
	$\forall 0 \leq l \leq i-1 \Rightarrow$
	$-aa[l]+1 \leq 0$
		$\forall k, k \neq i \Rightarrow 0 < aa[k]$

Shift [HKV11]
`a=0;`
`while(a<m){`
`aa[a+1]=aa[a];`
`a=a+1;`
`}`
	No invariants found.	$\forall k, k \neq i \Rightarrow aa[k] == aa[0]$

Sum Of Pairs [LRCR13]
`int m;`
`int *aa;`
`int x=getX(),`
`l=0, u=m-1;`
`//:$SORTED: aa @ASC`
`while (l < u) {`
`if (aa[l] + aa[u] < x)`
`l = l+1;`
`else`
`if (aa[l] + aa[u] > x)`
`u = u-1;`
`else break;`
`}`
	$\forall a, 0 \leq a \leq l-1 \Rightarrow$
	$aa[a] + aa[u] - x + 1 \leq 0$
		$\forall k, k \neq i - 2 \Rightarrow$
		$aa[k] + aa[u] < x$

Sequential Initialization [LRCR13]
`int main()`
`{`
`int m;`
`int *aa;`
`aa[0]=7;`
`int i=1;`
`while (i<m)`
`{`
`aa[i]=aa[i-1]+1;`
`++i;`
`}`
`}`
	$\forall x, 0 \leq x \leq i-2 \Rightarrow$
	$aa[x+1] - aa[x] - 1 \leq 0$
	$\forall x, 0 \leq x \leq i-2 \Rightarrow$
	$-aa[x+1] + aa[x] + 1 \leq 0$
		$\forall k, k \neq i - 2 \Rightarrow$
		$aa[k+1] == aa[k] + 1$

For the program *Initialisation* we see that both Cpp-inv and Gin-pink got the desired invariant. Gin-pink had as postcondition the formula $\forall k, 0 \leq k < m \Rightarrow aa[k] = 0$. The semantics of both invariants obtained is straightforward, with the observation that in Gin-pinks case $k \neq i$ has the meaning that for all values of k that were totally/partially processed.

The loop *Insertion* represents a program that takes a random element from an array and places it on the right the array of the array in such a way that all elements starting with the original position of the element and ending with one less than the new position. The invariant that we are looking for is the formula expressing that the value of the chosen element is less than than the values of the element between the original position and the new position of it. Both tools manage to infer this invariant. The form which is extracted by Cpp-Inv is as a inequality with one of the terms 0.

The postcondition we used for Gin-pink (translated in first order logic) is : $\forall k, k! = i - j - 2, aa[i - k] \geq x + 1$.

The loop *Partition* has as input an array *aa* which is partitioned in array *bb* for the non-negative elements and array *cc* for the negative elements. While Cpp-inv was able to find invariants characterizing the elements in *cc* as being negative and those in *bb* as non-negative, Gin-pink could get the invariant that every element in *bb* is equal to one in *aa*, whit the input postcondition $\forall k, k \neq b \Rightarrow 0 \leq B[k]$.

The loop $Maximum$ processes an array by comparing systematically the value of max with the value of every element in the array and assigning the value of the element in the current step to max if max is smaller. At the end of the loop the value of max is the largest value in the array. The invariant inferred by Cpp-inv expresses actually the property that for all elements that were processed, the difference between them and max is at most 0. This property is equivalent with the invariant of interest expressing that max is greater or equal than all elements that were processed. In the case of Gin-Pink, there was necessary a formula expressing that there is a total order on the type of the array. There is also an extra function that takes as input the limits of an array, the array and a variable m and returns $true$ if m is greater or equal than all elements in the array or $false$ otherwise.The invariant written in the table above expresses the fact that variable max is greater or equal to all elements in the array, which is the invariant we are looking for in this case.

In the case of $HeapProperty$ the loop checks until which position does the array has the heap property (every element has a value greater than the value of its parent) with the root in $aa[0]$, and such that the element at position i (except the leaves) has children at positions $2 * i + 1$ and $2 * i + 2$ (taking into account that the array has $2 * m$ elements). Cpp-Inv infer that invariant in the form that the difference between the "parent" and each of the "children" is less than 0. Also it infers a property expressing the fact that the values of the array grow faster than the value of the indexes. Gin-pink also managed to infer the property having the postcondition $forallk : int :: k! = i ==> A[k] < A[2 * k + 1]$. It also infers that for the value at position i all values that are past the position $2k$ are greater.

The loop named $FirstOccurence$ searches in a divide and conquer manner a value in a sorted array. Cpp-Inv finds finds two important invariants for understanding the loops: that all elements that are placed before the lower bound (l) are smaller than the value searched, and all elements that are laced before the location u is greater or equal with the value searched. From

this two properties on can infer that the value searched can be only between the locations l and u. Gin-pink can also infer these properties but the properties outputted by this tool are harder to understand and less explicit. The properties inferred by Gin-pink express only the fact that the value was not found yet through the elements that were already analyzed.

The *Palindrome* loop checks if an array is a palindrome (i.e. regardless of the way one reads it, it would be the same word). Cpp-inv is able to infer the formula expressing the property of a palindrome, by inferring two invariants in the following manner: restricting the values of a between 0 and i both the difference between $aa[m-a-1]-aa[a]$ and $aa[a]-aa[m-a-1]$ are less or equal to 0. Gin-pink inferred the property that all elements in the array that were already analyzed holds $aa[a] = aa[m-a]$, where a is a location already processed.

The *PartitionInit* loop compares the elements of two arrays aa and bb and keeps in array cc the locations at which the element in aa equals the one in bb. Cpp-inv infers the invariant $\forall l, 0 \leq l \leq c-1 \Rightarrow l - cc[l] \leq 0$ which can be interpreted in different ways, either that the size of cc is at most the size of aa, or that there can be at most elements in cc as there are in aa. The other invariants that Cpp-inv inferred is $\forall l, 0 \leq l \leq c-1 \Rightarrow cc[l] - i + c - l \leq 0$, which describes the property that i grows faster than c, much faster than l with respect to $cc[l]$. The second property is also inferred by Gin-pink.

The $Vararg$ is a loop that runs as long as the elements in the array aa are greater than 0. In the loop body there is no modification of the array, only the index is increased. Cpp-inv manages to infer that all elements processed are greater than 0. Gin-pink can also infer the invariant having the postcondition $\forall k, 0 \leq k < i \Rightarrow 0 < A[k]$.

The loop $Shift$ gives to every element, starting with the second, to take the value of the previous element. At the end of the loop all elements have the value of the initial value of the first one. For this loop Cpp-Inv does not find any invariant for this piece of code. The possible reason for which this happens is because the invariant does not fall into the template required by this method in order to infer it. Gin-pink uses the postcondition weakening successfully in order to prove the invariant that expresses that at all the processed locations there are values for the elements equal to the value of the first element.

The loop $SumofPairs$ has as input an increasingly ordered array and makes sure that the sum between the elements at location l and u is smaller than same random value x, and either increases l or decreases u, accordingly.

Cpp-inv infers the formula expressing that the sum between all values at positions that are between 0 and l and the value in the array at position u minus the value of x plus 1 is smaller or equal to 0. The same invariant is inferred by Gin-pink (in a different form) by weakening the postcondition $\forall k \neq l \Rightarrow aa[k] + aa[u] < x$.

SequantialInitialisation has as input an array with the first element initialized and processed by initializing the rest of the array with values with one higher than the value of the previous element. At the end of the loop the elements of the array represent a series of consecutive numbers. Cpp-inv manages to infer the invariant expressing this property in the form of two inequalities. Gin-pink can also infer the invariant with the postcondition $\forall k, k \neq i - 2 \Rightarrow aa[k + 1] = aa[k] + 1$. Although the postcondition is strong the tool still has to check that the invariant holds for the loop.

4.2 Experiments with Lingva

The table below presents the results obtained by running Lingva on the same set of loops as the other two tools. On the first column are written the loops used for this experiment, on the second column there is written one invariant of interest for the corresponding loop and, on the third column there is either an invariant obtained by Lingva together with the set of axioms and theorems that were used to reach to the result, or a different invariant (than the one on the second column) that could be of interest for the user. In the third column, the name of a variable followed by 0 represents the initial value of the variable.

At the end of the subsection we give explanations regarding the proof obtained and the theorems used in order to obtain the invariants.

program	Invariant of interest	Lingva

Initialisation
a=0;
while (a<m) do
aa[a]=0; a=a+1;
end do
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$aa[a] = 0$
		Refutation found.
		
		$\forall a, 0 \leq a \leq i-1 \Rightarrow$
		$aa[a] = 0$
		Theorems:
		6169. $false$
		535. $aa(sK0) \neq 0$
		382. $0 \leq sK0 \wedge \neg a \leq sK0 \wedge aa(sK0) \neq 0 \wedge$
		$aa(sK0) \neq aa0(sK0)$
		381. $\exists X0, ((0 \leq X0) \wedge \neg(a \leq X0) \wedge$
		$aa(X0) \neq 0 \wedge aa(X0) \neq aa0(X0))$
		380. $\exists X0, (((0 \leq X0) \wedge \neg(a \leq X0)) \wedge$
		$(aa(X0) \leq 0 \wedge aa(X0) \neq aa0(X0)))$
		379. $\neg\forall X0, (((0 \leq X0) \wedge \neg(a \leq X0)) \Rightarrow$
		$(aa(X0) = 0 \vee aa(X0) = aa0(X0)))$
		153. $\neg\forall X37, (((0 \leq X37) \wedge \neg(a \leq X37)) \Rightarrow$
		$(aa(X37) = 0 \vee aa(X37) = aa0(X37)))$
		152. $\neg\forall X37, (((0 \leq X37) \wedge (X37 < a)) \Rightarrow$
		$(aa(X37) = 0 \vee aa(X37) = aa0(X37)))$
		151. $\forall X37, (((0 \leq X37) \wedge (X37 < a)) \Rightarrow$
		$(aa(X37) = 0 \vee aa(X37) = aa0(X37)))$
		6168. $aa(sK0) = 0$
		6167. $aa(sK0) \neq aa(sK0) \vee aa(sK0) = 0$
		1567. $aa(X0) = aa0(X0) \vee aa(X0) = 0$
		1566. $aa(X0) = 0 \vee aa(X0) = aa0(X0) \vee$
		$aa(X0) = aa0(X0)$
		385. $(a0 + sK0(X0)) = X0 \vee aa(X0) = aa0(X0)$
		169. $\forall X0, (aa(X0) = aa0(X0) \vee$
		$(a0 + sK0(X0)) = X0)$
		3. $\forall X2, (aa(X2) = aa0(X2) \vee$
		$(a0 + sK0(X2)) = X2)$
		538. $aa((a0 + sK0(X0))) = 0 \vee$
		$aa(X0) = aa0(X0)$
		537. $aa(X0) = aa0(X0) \vee aa(X1) = 0 \vee$
		$(a0 + sK0(X0)) \neq X1$
		388. $aa(X0) = aa0(X0) \vee aa(X1) = X2 \vee$
		$0 \neq X2 \vee (a0 + sK0(X0)) \neq X1$
		172. $\forall X0, X1, X2, ((a0 + sK0(X0)) \neq X1 \vee$
		$0 \neq X2 \vee aa(X1) = X2 \vee aa(X0) = aa0(X0))$
		6. $\forall X2, X0, X1, ((a0 + sK0(X2)) \neq X0 \vee$
		$0 \neq X1 \vee aa(X0) = X1 \vee aa(X2) = aa0(X2))$
		536. $aa(sK0) \neq aa0(sK0)$

Insertion
x=aa[i];
j = i-1;
while (j >= 0 and
aa[j] > x) do
aa[j+1] = aa[j];
--j;
end do
	$\forall a, a \leq 0 \wedge$
	$a \leq i-j-2 \Rightarrow$
	$x+1 \leq aa[-a+i]$
		Refutation not found.
		
		Other interesting
		properties found :
		$\forall X0,: (sK0(X0) \leq (j-j0) \vee$
		$aa(X0) = aa0(X0))).$
		$\forall X0, X1 : ((1+j0) \neq X0 \mid$
		$aa0(j0) \neq X1 \mid aa(X0) = X1 \mid$
		$((j-j0) \leq 0)).$

Partition
a=0;
b=0;
c=0;
while (a<m) do
if(aa[a]>=0)
then bb[b]=aa[a];
b=b+1;
else cc[c]=aa[a];
c=c+1;
end if;
a=a+1;
end do
	$\forall a, 0 \leq a \leq c-1 \Rightarrow$
	$cc[a] < 0$
	$\forall a, 0 \leq a \leq b-1 \Rightarrow$
	$bb[a] \geq 0$
		Refutation found
		
		1224. *false* (1:0)
		[resolution 314,968]
		968. $\neg(c \leq X1)$ (0:3)
		[cnf transformation 661]
		661. $\forall X1 : (0 \leq X1) \wedge \neg(c \leq X1))$
		$\wedge(0 \leq bb(sK0))$ [skolemisation 660]
		660. $\exists X0 : (\forall X1 : ((0 \leq X1) \wedge$
		$\neg(c \leq X1)) \wedge (0 \leq bb(X0)))$
		[ennf transformation 659]
		659. $\neg\forall X0 : (\forall X1 : ((0 \leq X1) \wedge$
		$\neg(c \leq X1)) \Rightarrow \neg(0 \leq bb(X0)))$
		308. $\neg\forall X68 : (\forall X68 : ((0 \leq X68) \wedge$
		$\neg(c \leq X68)) \Rightarrow \neg(0 \leq bb(X68)))$
		307. $\neg\forall X68 : (\forall X68 : ((0 \leq X68) \wedge$
		$(X68 < c)) \Rightarrow (bb(X68) < 0))$
		[negated conjecture 306]
		306. $\forall X68 : ((0 \leq X68) \wedge (X68 < c)) \Rightarrow$
		$(bb(X68) < 0)$ [input implication]
		314. $(X0 \leq X0)$ (0:3) [theory axiom]

Maximum
int i=1;
int max = aa[0];
while (i<m)
{
if (max<aa[i]) {
max = aa[i];
}
++i;
}
	$\forall a, 0 \leq a < i \Rightarrow$
	$aa[a] \leq max$
		Refutation not found.
		
		Other interesting
		properties that were
		proven:
		$\forall X \exists Y, 0 \leq X, Y < i \Rightarrow aa[X] \leq aa[Y]$
		
		Theorems:
		
		904. *false* [resolution 159,529]
		529. $\neg(i \leq X0)$
		[cnf transformation 375]
		375. $\forall X0, ((0 \leq sK2(X0)) \wedge$
		$\neg(i \leq sK2(X0)) \wedge (0 \leq X0) \wedge$
		$\neg(i \leq X0) \wedge \neg(aa(sK2(X0)) \leq aa(X0)))$
		[skolemisation 372]
		372. $\forall X0, \exists X1((0 \leq X1) \wedge$
		$\neg(i \leq X1) \wedge (0 \leq X0) \wedge$
		$\neg(i \leq X0) \wedge \neg(aa(X1) \leq aa(X0)))$
		[flattening 371]
		371. $\forall X0, \exists X1(((0 \leq X1) \wedge$
		$\neg(i \leq X1) \wedge (0 \leq X0) \wedge$
		$\neg(i \leq X0)) \wedge \neg(aa(X1) \leq aa(X0)))$
		368. $\neg\exists X0, \forall X1((0 \leq X1) \wedge$
		$\neg(i \leq X1) \wedge (0 \leq X0) \wedge \neg(i \leq X0) \Rightarrow$
		$(aa(X1) \leq aa(X0)))$
		153. $\neg\exists X37, \forall X36((0 \leq X36) \wedge$
		$\neg(i \leq X36) \wedge (0 \leq X37) \wedge \neg(i \leq X37) \Rightarrow$
		$(aa(X36) \leq aa(X37)))$
		150. $\neg\exists X37, \forall X36((0 \leq X36) \wedge$
		$(X36 < i) \wedge (0 \leq X37) \wedge (X37 < i) \Rightarrow$
		$(aa(X36) \leq aa(X37)))$
		149. $\exists X37, \forall X36((0 \leq X36) \wedge$
		$(X36 < i) \wedge (0 \leq X37) \wedge (X37 < i) \Rightarrow$
		$(aa(X36) \leq aa(X37)))$
		159. $(X0 \leq X0)$ (0:3) [theory axiom]

HeapProperty
const int m;
assume(m>=0);
int aa[2*m];
int i=0;
while (2*i+2<2*m)
{
if (aa[i]>aa[2*i+1] or
aa[i]>aa[2*i+2])
break;
++i;
}
	$\forall 0 \leq a \leq i-1$
	$aa[a] \leq aa[2a]$
	$2a - aa[2a+1] + aa[a]$
	$-2m + 3 \leq 0$
	$-aa[a+2] + aa[a] \leq 0$
	$2a - aa[a+2] + aa[a]$
	$-2m + 3 \leq 0$
		Can not be checked
		due to the structure
		of the program.

FirstOccurence
int aa[m];
int x = readX();
int l = 0;
int u = m;
 	
//aa is sorted asscendent
while (l < u) {
int m = (l+u)/2;
if (aa[m] < x) l = m+1;
else u = m;
}
	$\forall a, 0 \leq a \leq m - U - 1 \Rightarrow$
	$x \leq aa[m-a-1];$
	$aa[u+x] \leq aa[m-a-1] -$
	$-2m + 2l;$
	$\forall a, 0 \leq a \leq l-1 \Rightarrow$
	$aa[a] \leq x - 1;$
	$aa[a] \leq 4l - 2u + x - 3;$
		Can not be checked
		due to
		the use of divide.

Palindrome
int aa[m];
int i=0;
while (i<m/2)
{
if (aa[i] != aa[m-i-1])
{
break;
}
i++;
}
	$\forall a, 0 \leq a \leq i-1 \Rightarrow$
	$aa[m-a] = aa[a]$
		Can not be checked.
		The \&\&-logical
		operator is
		not supported

PartitionInit
int aa[m], bb[m], cc[m];
int i=0, c=0;
while (i<m)
{
if (aa[i] == bb[i])
{
cc[c] = i;
c++;
}
i++;
}
}
	$\forall X, 0 \leq x \leq c-1 \Rightarrow$
	$aa[cc[x]] = bb[cc[X]]$
		Refutation not found.
		Other interesting
		properties that
		were proved:
		$\forall X3, X4, X5$
		$C0 + X3 = X4 \wedge$
		$a0 + X3 = X5 \wedge$
		$0 \leq X3 \Rightarrow$
		$C[X4] = X5 \vee$
		$c - c0 \leq X3$

Vararg
a=0;
while(aa[a]>0){
a=a+1;
}
	$\forall x, 0 \leq x \leq a-1 \Rightarrow$
	$aa[x] > 0$
		Refutation found.
		$\forall x, 0 \leq x \leq a-1 \Rightarrow$
		$aa[x] > 0$
		
		Theorems:
		1530. $false$ (2:0)
		[subsumption resolution 1529,574]
		574. $\neg(a \leq (a0 + sK0))$
		[forward demodulation 491,142]
		142. $(X0 + X1) = (X1 + X0)$
		[theory axiom]
		491. $\neg(a \leq (sK0 + a0))$
		[cnf transformation 350]
		350. $(0 \leq sK0) \wedge \neg(0 \leq aa((a0 + sK0))) \wedge$
		$\neg(a \leq (sK0 + a0))$ [skolemisation 349]
		349. $\exists X0, ((0 \leq X0) \wedge \neg(0 \leq aa((a0 + X0))) \wedge$
		$\neg(a \leq (X0 + a0)))$ [flattening 348]
		348. $\exists X0, (((0 \leq X0) \wedge \neg(0 \leq aa((a0 + X0)))) \wedge$
		$\neg(a \leq (X0 + a0)))$
		[ennf transformation 141]
		141. $\neg\forall X0, (((0 \leq X0) \wedge$
		$\neg(0 \leq aa((a0 + X0)))) \Rightarrow (a \leq (X0 + a0)))$
		[evaluation 140]
		140. $\neg\forall X0, (((0 \leq X0) \wedge$
		$(aa((a0 + X0)) < 0)) \Rightarrow (a \leq (X0 + a0)))$
		[negated conjecture 139]
		139. $\forall X0, (((0 \leq X0) \wedge$
		$(aa((a0 + X0)) < 0)) \Rightarrow (a \leq (X0 + a0)))$
		[input implication]
		1529.$(a \leq (a0 + sK0))$
		[forward demodulation 1528,142]
		1528.$(a \leq (sK0 + a0))$
		[subsumption resolution 1518,489]
		489. $(0 \leq sK0)$ [cnf transformation 350]
		1518. $\neg(0 \leq sK0) \vee (a \leq (sK0 + a0))$
		[resolution 355,586]
		586. $(aa((a0 + sK0)) \leq 0)$
		[resolution 149,490]
		490. $\neg(0 \leq aa((a0 + sK0)))$
		[cnf transformation 350]
		149. $(X0 \leq X1) \vee (X1 \leq X0)$
		[theory axiom]
		355. $\neg(aa((a0 + X0)) \leq 0) \vee \neg(0 \leq X0)$
		$\vee(a \leq (X0 + a0))$ [cnf transformation 158]
		158. $\forall X0, ((a \leq (X0 + a0)) \vee$
		$\neg(0 \leq X0) \vee \neg(aa((a0 + X0)) \leq 0))$
		[flattening 5]
		5. $\forall X0, ((a \leq (X0 + a0)) \vee$
		$\neg(0 \leq X0) \vee \neg(aa((a0 + X0)) \leq 0))$
		[input inv4]

Shift
a=0;
while(a<m){
aa[a+1]=aa[a];
a=a+1;
}
	$\forall x, 0 \leq x < a \Rightarrow$
	$aa[x] = aa[0]$
		Refutation found.
		$\forall X, 0 \leq X \wedge X < a \Rightarrow$
		
		$aa[X] = aa[X+1]$
		Theorems used:
		1247. *false* (1:0) [resolution 1096,530]
		530. $\neg(a \leq sK0)$ (0:3) [cnf transformation 379]
		379. $\forall[X1] : (0 \leq X1) \wedge \neg(a \leq sK0) \wedge$
		$aa(sk0) \neq aa(sk0+1)$ [skolemisation 378]
		378. $\exists[X0] : (\forall[X1] : (0 \leq X1) \wedge$
		$\neg(a \leq X0) \wedge aa(X0) \neq aa(X0+1))$ [flattening 377]
		377. $\exists[X0] : ((\forall[X1] : (0 \leq X1) \wedge \neg(a \leq X0)) \wedge$
		$aa(X0) \neq aa(X0+1))$ [ennf transformation 376]
		376. $\neg\forall[X0] : ((\forall[X1] : (0 \leq X1) \wedge \neg(a \leq X0)) \Rightarrow$
		$aa(X0) = aa(X0+1))$ [rectify 152]
		152. $\neg\forall[X36] : ((\forall[X36] : (0 \leq X36) \wedge \neg(a \leq X36)) \Rightarrow$
		$aa(X36) = aa(X36+1))$ [evaluation 151]
		151. $\neg\forall[X36] : ((\forall[X36] : (0 \leq X36) \wedge (X3 < a)) \Rightarrow$
		$aa(X36) = aa(X36+1))$ [negated conjecture 150]
		150. $(\forall[X36] : (0 \leq X36) \wedge (X36 < a)) \Rightarrow$
		$aa(X36) = aa(X36+1)$ [input implication]
		1096. $(a \leq X0)$ (0:3)
		[subsumption resolution 925,878]
		878. $(X3 \leq (X3+X2))$ (3:5)
		[superposition 853,153]
		153. $(X0+X1) = (X1+X0)$ (0:7) [theory axiom]
		853. $(X11 \leq (X12+X11))$ (2:5)
		[subsumption resolution 843,529]
		529. $(0 \leq X1)$ (0:3) [cnf transformation 379]
		843. $(X11 \leq (X12+X11)) \vee \neg(0 \leq X12)$
		(2:8) [superposition 161,669]
		669. $(0+X0) = X0$ (1:5) [superposition 153,155]
		155. $(X0+0) = X0$ (0:5) [theory axiom]
		161. $((X0+X2) \leq (X1+X2)) \vee \neg(X0 \leq X1)$
		(0:10) [theory axiom]
		925. $\neg(a0 \leq (a0+(0+X0))) \vee (a \leq X0)$
		(0:10) [backward demodulation 897,491]
		491. $(a \leq X0) \vee \neg(a0 \leq (a0+(-a+X0)))$
		(0:11) [cnf transformation 337]
		337. $\forall[X0] : (\neg(a0 \leq (a0+(-(a)+X0))) \vee (a \leq X0))$
		[flattening 336]
		336. $\forall[X0] : (\neg(a0 \leq (a0+(-(a)+X0))) \vee (a \leq X0))$
		[rectify 112]
		112. $\forall[X1] : (\neg(a0 \leq (a0+(-(a)+X1))) \vee (a \leq X1))$
		[input inv111]
		897. $0 = -(X0)$ (4:4)
		[subsumption resolution 895,529]
		895. $\neg(0 \leq -(X0)) \vee 0 = -(X0)$
		(4:8) [resolution 877,164]
		164. $\neg(X1 \leq X0) \vee \neg(X0 \leq X1) \vee X0 = X1$
		(0:9) [theory axiom]
		877. $(-(X1) \leq 0)$ (3:4) [superposition 853,157]
		157. $(X0+(-X0)) = 0$ (0:6) [theory axiom]

Sum Of Pairs
`int m;`
`int *aa;`
`int x=getX(),`
`l=0, u=m-1;`
 	
`//:$SORTED: A @ASC`
`while (l < u) {`
`if (aa[l] + aa[u] < x)`
`l = l+1;`
`else`
`if (aa[l] + aa[u] > x)`
`u = u-1;`
`else break;`
`}`
	$\forall x, 0 \leq x \leq m-u-2 \Rightarrow$
	$x+1 \leq A[m-x-1]+A[l]$
		Refutation not found.

Sequential Initialisation
`int main()`
`{`
`int m;`
`int *aa;`
 	
`aa[0]=7;`
`int i=1;`
`while (i<m)`
`{`
`aa[i]=aa[i-1]+1;`
`++i;`
`}`
`}`
	$\forall x, 0 \leq x \leq i \Rightarrow$
	$aa[x+1] = aa[x]+1$
		timelimit reached at 200
		Other interesting property that was proven:
		$i - i0 < sk0(X0) \Rightarrow$
		$aa(X0) = aa0(X0)$
		$m \leq i0 + sk0(X1) \Rightarrow$
		$aa(X1) = aa0(X1)$
		$sk0(X2) \leq 0 \Rightarrow$
		$aa(X2) = aa0(X2)$
		$m \leq i + i0 \wedge 0 \leq i \Rightarrow$
		$0 \leq i0$

For the loop *Initialisation* first vampire proves that $aa(sk0) = 0(6168)$ and afterwards that $aa(sk0)! = 0(535)$. We explain the proof in three steps:

1. the formula representing the invariant of interest being negated and after a series of transformations, evaluation, rectify, flattening and skolemisation, the negated formula is put in a standard form

   $\$lesseq(0, sK0)\& \$lesseq(a, sK0)\&aa(sK0)! = 0\&aa(sK0)! = aa0(sK0)$ (382). From this formula it is easy to see that formula 6168 can be deduced

2. For the second formula (535) the proof starts with on of the formulas that *lingva* inferred $![X2, X0, X1] : (\$sum(a0, sK0(X2))! = X0|0! = X1|aa(X0) = X1|aa(X2) = aa0(X2))(6)$. After cnf transformation and equality resolution the following formula is obtained $aa(\$sum(a0, sK0(X0))) = 0|aa(X0) = aa0(X0)(538)$.

3. Using another formula (after processing it) that is automatically deduced by *Lingva*, $\$sum(a0, sK0(X0)) = X0|aa(X0) = aa0(X0)$, *Vampire* uses superposition that has as result $aa(X0) = aa0(X0)|$ $aa(X0) = 0(1567)$, after eliminating the duplicate literals. From

the formula (382) also used in the first step *Vampire* infers also $aa(sK0)! = aa0(sK0)(536)$. Using superposition on (536) and (1567) and after eliminating trivial inequalities $aa(sk0) = 0$ results.

For the loop *Partition* two invariants were proved by *Vampire*, one formulating that for every non-negative element in *aa* there is an element in *bb*, and one formulating that for every negative element in *aa* there is an element in *cc*. We are going to explain only the former proof because of the similarities between the two. From the input implication after processing it *vampire* infers the clause $\$lesseq(0, X1)(982)$. Vampire use the property obtained by *Lingva* that expresses the fact that the iterator *c* grows as most as fast as iterator *a*, $![X66, X67] : (\$lesseq(X66, \$sum(c, X67))| \$lesseq(X66, c0)|$ $\$lesseq(0, \$sum(a, \$sum(\$uminus(a0), X67))))(1429)$. Using resolution on (1429) and (982) the formula $\$lesseq(X0, \$sum(c, X1))| \$lesseq(X0, c)$ (1447) is inferred. Using the following three theory axioms $\$sum(X0, X1) = \$sum(X1, X0)$, $\$lesseq(\$sum(X0, 1), X1)| \$lesseq(X1, X0)$, $\$lesseq(X0, X0)$ and the previous inferred formula and the theory of superposition the following formula is inferred $\$lesseq(\$sum(1, \$sum(0, X4)), 0)$ (1714). From the formula tagged with (1714) and the theory axiom $\$sum(0, X0) = X0$ the following formula is inferred $\$lesseq(\$sum(1, X4), 0)$. The contradiction is inferred from the las formula and the axiom that every number summed with its opposite will result to zero. This proof makes use of a large number of theory axioms together with input axioms discovered by *Lingva* and because of this fact its complexity is higher than that of other discussed proofs. The invariant that expresses the property for array *cc* has a similar proof with corresponding differences where these apply.

For the loop *Insertion* we try to prove the inductive formula representing the invariant. Since *Vampire* has not yet a mechanism that deals with induction this invariant is harder to prove. On the other hand *Lingva* automatically infers two properties that are necessary in understanding the program. The first one expresses the fact that from a certain position the elements of the array do not modify their value. The second property automatically inferred shows that the invariant holds for the first value of j. In the next chapter we show that by adding the right axioms to the ones already outputted by *Lingva* the theorem prover can find a proof for the searched invariant.

For the loop *Maximum*, although the invariant we were looking for could not be proven *Vampire* proved that for every iteration of the loop there exists an element that was already processed and that is greater or equal to all other processed elements. *Vampire* proves this by simply getting a contradiction with the axiom $X \leq X$. In the next chapter we show what

properties can be added in order to prove the invariant of interest.

For the moment *Lingva* has some limitation that do not permit the analysis of loops that have more than one condition. Also the analysis of statements that use division is impossible. Even if the statements are modified in such a way that multiplication is used instead of division the analysis of the statements is still troublesome.For the three loops $HeapProperty$, $FirstOccurence$, $Palindrome$, the properties automatically inferred do not offer information about array elements, but rather information about the iterators that could not be used in order to prove any invariants of interest. Because of the current limitations in the analysis system of *Lingva* we consider that these three loops could not be representative for the capabilities of proving invariants.

For the loop $PartitionInit$ the invariant that we are interested in is also an inductive invariant. As we specified before, this type of invariant is, for $Vampire$, harder to prove. *Lingva* managed to automatically infer a different invariant expressing the property that every elements in the array *cc* is a valid position in the array *aa*. There are no properties inferred regarding the array *bb*, so in order to infer the invariant of interest there are other properties that have to be added to the list of formulas outputted by *Lingva*.

For the loop *vararg* the invariant is proved by $Vampire$ only based on the input obtained from *Lingva*. With a few transformation from the (fourth) input invariant the following formula can be obtained:
$\$lesseq(aa(\$sum(a0, X0)), 0)| \ \$lesseq(0, X0)|\$lesseq(a, \$sum(X0, a0))$ (355). This property can be rewritten into an implication expressing that if $X0$ is positive and the element at position $a0 + X0$ is also positive then the position $a0 + X0$ is smaller than a. After transforming the formula representing the invariant $Vampire$ inferred three new clauses
$\neg\$lesseq(0, aa(\$sum(a0, sK0)))$ (490), $\$lesseq(0, sK0)$ (350) and
$\$lesseq(a, \$sum(sK0, a0))$ (491). The formula (490) can be transformed in the formula
$\$lesseq(aa(\$sum(a0, X0)), 0)$ (586) using resolution and a theory axiom. Using resolution a few times on this formula $Vampire$ infers the following formula $\$lesseq(a, \$sum(a0, sK0))$ (1529). Using forward demodulation on (491) an the theory axiom
$\$sum(X0, X1) = \$sum(X1, X0)$ and get the formula $\neg\$lesseq(a, \$sum(a0, sK0))$ which is the opposite of formula (1529).

For the loop $Shift$ the proof of refutation for the invariant of interest was found. We give a sketch of the proof for easier understanding of it. From the negated conjuncture the clause $\neg(a \leq sk0)$ (530) can be inferred. This

clause is in contradiction with clause $a \leq X0$ (1096) which is inferred as follows. From the negated conjunction together with theory axioms one can infer $X11 \leq X12 + X11 \vee \neg(0 \leq X12)$ (853). From theory axioms and negate input invariant one can also infer $0 = -X0$ (897). From the previous clause and the input invariant

$\forall[X1] : (\neg(a0 \leq (a0 + (-(a) + X1))) \vee (a \leq X1)$ the clause $\neg(a0 \leq (a0+(0+X0))) \vee (a \leq X0)(925)$ can be inferred. From the latter formula and formula labeled with (853) one can infer the clause labeled (1096) making the refutation proof complete.

For the loop $SumofPairs$ the invariant that we are trying to prove is based on a property that is not explicitly expressed in the program. The fact that the input array is ordered has a great impact in the proving power of the theorem prover. In the next chapter we show how the addition of this property makes possible the proof of the invariant of interest.

For the loop $SequentialInitialisation$ the invariant of interest expresses the property that the elements of the array have consecutive values. Although for this specific loop *Lingva* discovered many invariants that could be used in order to get a proof of the invariant of interest, the limitations of the machine did not make this possible, the computations being too slow. Nevertheless we introduce new theorems to the already formed set in order to get faster results. This situation is also discussed in the next chapter. From the direct output of *Lingva* we can distinguish a few properties that give understanding with respect to the loop. More precisely it shows that if a position was not visited yet, the value of the element did not change.

4.3 Discussions of Experimental Results

From the experiments above we can draw a few conclusions about the invariants that these tools can infer. $Cpp - inv$ has very good results with most of the experiments. The running time necessary to infer the invariants is appreciable even when run on a machine with not too many resources. A lot of invariants studied fell into the template that this tool is using, being able to automatically find them without any guidance from the user. It can also analyze more complex loops that have in the body $if \ldots then \ldots else$ statements and loops with more than one condition which gives better flexibility and expressibility when writing the code to be analyzed. One important class of invariants that this tool is able to infer is the induction type. This type of properties usually indicate a relation between an element of the array and the element or elements prior to it.

In the benchmark used we found a single loop that could not be analyzed properly by this tool, $Shift$. The invariant we were looking for was a relation between an element at a certain position i and the element at position

$i + 1$. Since this property needed reasoning about a position that was not reached yet, this did not meet the requirements for the template used by $Cpp{-}inv$. For the invariant $PartitionInit$ the invariant that were discovered did not give insight on the relation between the elements of the three arrays processed in the loop. All other invariants that were concluded for the respective loops were the invariant of interest. The tool was able to extract useful information about the usage of the array in the loop and yield formulas of great usage in order for the user to understand the task done in the snippet of code.

What seems to be the drawback of this method is the fact that one can not independently use it to check a certain property of interest if it is not already inferred by the tool. One can not add new information that can not be automatically inferred in order to get a result about the property of interest. On the other hand if a user decides to check for properties a piece of code seen for the first time, this tool might give useful information regarding the elements of the processed array.

In the case of $Gin - pink$ there is a large set of invariants that can be inferred from the postcondition and proved by $Boogie$. The advantage of using postconditions is that the invariants that will be derived from them are usually useful for understanding the program. Since the program has as starting point for searching an invariant a property that is of interest for the user it is natural that the invariant found by weakening the formula would also be of interesting when the code is analyzed by the user. On the other hand it might the case that the invariant would contain auxiliary variables that are not part of the postcondition. In this case the invariant of interest is not going to be derived by the tool.

Although $Gin - pink$ is not able to infer (without any interference from the user) an invariant containing auxiliary variables with respect to the postcondition there are walk-arounds this problems. One variant to work this problem out is to augment postconditions with one clause that specifies properties of the auxiliary variable. At this point $Gin - pink$ might have a chance to derive the wanted invariant by weakening the new postcondition. One of this loops is the one tagged with $Maximum$ which was modified in order to see if the tool would successfully infer the invariant we were looking for.

Another type of invariants that cause problems to this tool is the one that include product operation. In this case the problem comes from the theorem prover that is embedded in the tool, namely $Z3$. This theorem prover does not have the background theory necessary to handle formulas that have product of numeric in their composition. Unfortunately for this case there is no workaround to make the tool infer the invariant since there is no way to include new theories that could be used for processing such formulas.

From the experiments seen above we can see that there are a lot of

useful invariants derived by this tool, and making use of the method that helps derive invariants with auxiliary variables would give the user better results.

For the tool *Lingva* we observe from the experiments that it obtains a lot of properties from analyzing the code. To get an idea of how precise the code is analyzed, the number of invariants that are outputted after eliminating the trivial ones is somewhere around 150, for a piece of code with 10 lines. Also the invariants that are inferred can also express properties between different variables and also between initial values and new values of the variables, feature that was not present for the other two tools.

The drawback of this method is that some loops that contain more that have more than one condition to be checked can not be analyzed because the tool has not yet implemented the analysis of && and || operators from C programming language. This problem can be solved if the loop does not contain any $if \ldots then \ldots else$ statements, because at the moment the loop can be successfully analyzed if it is not nested and does not contain nested if statements.

An operation from which is hard to infer invariants is the product over integer. This method is able to infer such invariants from the direct analysis of the loop. In order to see the efficiency of this feature of the tool we modified some of the above loops and processed them with *Lingva*. Although the invariants obtained were expressing properties that need the multiplication operation we could not get any interesting invariants (since also the loops were not have a real purpose in processing an array).

In the set of loops that we studied there was a significant number that had as invariants of interest invariants that needed induction in order to be proven. Unfortunately the theory of induction is not yet implemented in *Vampire* so even though the invariants inferred by *Lingva* were enough to prove them this did not happen. In the next section we show how we can nevertheless infer this invariants if we study the properties deduced by *Lingva* and add the missing properties in order for the induction to be complete.

This method has the advantage that if the user knows some information about the analyzed loop, or any theory that might not be implemented in *Vampire* yet it can easily encode the formulas representing the rules in *Vampire* and run the theorem prover in order to see if a proof of refutation for the invariant of interest is possible, with the new information. We took advantage of this fact and use it to prove all the invariants that were not automatically proved. In the next chapter we show in detail how the proof of refutation were obtained for each invariant.

The three methods have different strong points, $Cpp - inv$ can infer induction properties without any supplementary information from the user, while for *Lingva* this is not a trivial task, although it is achievable. $Gin - pink$ can also deduce such invariants as long as the postcondition mentions

the variable used as an index, otherwise the task is impossible because it does not have a method to add information about other variables.

Lingva is the only one from the three tools that can infer properties about the initial values and the modified values of the variables, by introducing a new variable name for the initial value of the variable. This feature offers the user a more expressive way to reason about the program. It brings more information to the output as it can infer that starting with one position of the array the elements don't further change, information that can not be rendered by the other two tools.

Lingva is more flexible in the sens that knowing that some information can not be inferred from the loop (such as if the array is sorted) the use can add this information to the output of lingva in order to obtain a possibly better result from $Vampire$. $Gin - pink$ also accepts in the input new information from the user, in the form of *assume* statements, but in this case the type of formula that one can provide is not as expressible as first order logic.

$Gin - pink$ has a great advantage by being a goal oriented method. The invariants inferred by this tool are certainly of interest for the user, since they have as starting point the postcondition that is interesting for the program. From this point of view *Lingva* tries to infer as many invariants as possible, to make sure it covers as many properties that might be of interest as possible, while $Cpp-inv$ looks for properties that fall in a certain pattern, between specific point of the program analyzed.

5 Invariant Specific Theory Extensions to First Order Theorem Prover

In this chapter we proceed to analyze the programs for which the properties of interest could not be proven by Lingva/Vampire and try to find additional properties of the variables in these programs that we could add such that the computing power of the prover is increased.

5.1 Comparison of invariants strength

The Maximum Example. The first program to be analyzed is the one with the name "Maximum". In plain English what this program does is to put the value of the first element of an array *aa* into the variable *max* and iterate through the rest of the array comparing max with the rest of elements and changing the current value of *max* with the greater value of the elements if it is the case. The followings are the lines in the loop:

```
int i=1;
int max = aa[0];
while (i<m)
{
  if (max<aa[i]) {
    max = aa[i];
  }
  ++i;
}
```

Analyzing the output of *Lingva* we observe that the properties with respect to the iterator i are not strong enough so we add one to assure that the initial value of i is 0:

Property. 1 *tff(prop1,axiom, i0=0).*

and one to make ensure that i is increasing:

Property. 2 *tff(prop2,axiom, $less(i0,i)).*

These two axioms are necessary to ensure that the first part of the implication will not be invalidated by a false positive.

We also add two properties regarding the variable *max*. The first one ensures that the initial value for *max* is equal with the value of the first element in *aa*:

Property. 3 *tff(prop3, axiom, max0=aa(0)).*

While the second one ensures that *max* takes its values only from the values of elements in *aa*:

Property. 4 *tff(prop4, axiom, ?[Z:$int]: $lesseq(0,Z) & $lesseq(Z,n) & max=aa(Z)).*

Since the properties inferred by *Lingva* did not show any relation between the variable *max* and the elements in the array it is hard to infer such a property. The two axioms above have the role to make this connection in order for the reasoning to be possible.

With the above properties added to the output of *Lingva* the theorem prover can prove that all elements in the array that were already examined have the value at most the value of *max* expressed in the following formula:

Property. 5 *tff(implication, conjecture, ![X: $int]: ($lesseq(0,X) & $less(X,i) =>$lesseq(aa(X),max))).*

The proof of refutation outputted by *Vampire* is given in the following lines:

```
Refutation found. Thanks to Tanya!
981. $false (0:0) [subsumption resolution 980,906]
906. sP5 (1:1) [resolution 903,530]
530. $lesseq(X0,i) (0:3) [cnf transformation 374]
374. ! [X0] : ($lesseq(0,i) & $lesseq(0,sK2(X0)) &
$lesseq(sK2(X0),i)& $lesseq(0,X0) & $lesseq(X0,i) &
~$lesseq(aa(sK2(X0)),aa(X0))) [skolemisation 371]
150. ? [X37] : ! [X36] : (($lesseq(0,i) & $lesseq(0,X36) &
$lesseq(X36,i) & $lesseq(0,X37) & $lesseq(X37,i)) =>
$lesseq(aa(X36),aa(X37)))
[input implication]
903. ~$lesseq(1,i) | sP5 (0:4) [splitting component introduction]
980. ~sP5 (0:1) [subsumption resolution 953,529]
529. $lesseq(0,X0) (0:3) [cnf transformation 374]
953. ~$lesseq(0,$uminus(i)) | ~sP5 (0:5)
[backward demodulation 923,904]
904. ~$lesseq(0,$sum(0,$uminus(i))) | ~sP5 (0:7)
 [splitting 669,903]
669. ~$lesseq(1,i) | ~$lesseq(0,$sum(0,$uminus(i))) (0:9)
[definition unfolding 512,517,517]
517. i0 = 0 (0:3) [cnf transformation 143]
143. i0 = 0 [input prop1]
512. ~$lesseq(1,i) | ~$lesseq(i0,$sum(i0,$uminus(i))) (0:9)
[cnf transformation 357]
357. ~$lesseq(i0,$sum(i0,$uminus(i))) | ~$lesseq(1,i)
  [flattening 138]
138. ~$lesseq(i0,$sum(i0,$uminus(i))) | ~$lesseq(1,i)
  [input inv137]
923. $sum(0,X0) = X0 (1:5) [superposition 153,155]
```

```
155. $sum(X0,0) = X0 (0:5) [theory axiom]
153. $sum(X0,X1) = $sum(X1,X0) (0:7) [theory axiom]
```

From inv137 inferred by *Lingva* and the property 1 $Vampire$ deduces that either i is less or equal to 1 or -1 is smaller than 0. At proposition 903 a new splitting component in a disjunction with i smaller than 1. From this formula and the negation of the invariant $Vampire$ infers $sP5$. From the negated conjunction also the fact that $X0$ is greater than 0 is inferred and use this to deduce $sP5$. The refutation can now be completed.

Although *Vampire* could not automatically find a refutation proof for the formula representing the property of max, the extra-formulae that were added by hand was just information about the variables and not new theories.

The Partial Initialization Example. The second loop we are going to look at is " Partial Initialization ". This program has as input two arrays, *aa* and *bb*, and saves the indexes for which the element in *aa* equals the one in *bb* in the array *cc*.

```
 int aa[m], bb[m], cc[m];
  int i=0, c=0;
  while (i<m)
  {
    if (aa[i] == bb[i])
    {
      cc[c] = i;
      c++;
    }
    i++;
  }
}
```

We add properties that give information about the initial values of the iterators a and c:

Property. 6 *tff(prop2, axiom, a0=0).*

Property. 7 *tff(prop4, axiom, c0=0).*

The properties inferred by *Lingva* with respect to these iterators are too weak and it can not be inferred that all their values start with the same value, and that the value at which they start is non-negative.

We also limit the the execution of the programs to the situation when the number of elements in the array *aa*, m, is greater than 0:

Property. 8 *tff(prop5, axiom, $lesseq(0, m))*

This property is necessary because from the static analysis of the program it can not be inferred that the array has a positive number of elements, but we know that any other case would not make sense.

We also add a property that expresses the upper and lower limit of that an element in *cc* can get, and also a property that establish a relation between the elements in the array *cc* and the value of *c* for the corresponding element:

Property. 9 *tff(prop1, axiom, ![Y:$int]: $lesseq(0,cc(Y))& $less(cc(Y),m)).*

Property. 10 *tff(prop6, axiom, ![Y0:$int]:($lesseq(0,Y0)& $less(Y0,c)=> $lesseq(Y0,cc(Y0)))).*

With this new axioms added although *Vampire* can not prove the initial property: $\forall X, 0 \leq X \wedge X < c \Rightarrow aa[cc[X]] == bb[cc[X]]$, it can prove another formula with one quantifier alternation: $\forall X, \exists Y, 0 \leq X \wedge X < m \wedge 0 \leq Y \wedge Y < c \wedge aa(X) = bb(X) \Rightarrow cc(Y) = X$.

Here is the proof of refutation that was outputted by *Vampire*:

```
Refutation found. Thanks to Tanya!
2064. $false (1:0) [subsumption resolution 2061,2048]
2048. ~$lesseq(c,0) (1:3) [resolution 2019,1058]
1058. $lesseq(X1,c) (0:3) [cnf transformation 746]
746. ! [X1] : ($lesseq(0,X1) & $lesseq(X1,c) & bb(sK0) = aa(sK0)&
cc(X1) != sK0 & cc(X1) != cc0(X1)) [skolemisation 745]
310. ! [X48] : ? [X49] : (($lesseq(0,X49) & $lesseq(X49,c) &
bb(X48) = aa(X48)) => (cc(X49) = X48 | cc(X49) = cc0(X49)))
[input implication]
2019. ~$lesseq(1,c) | ~$lesseq(c,0) (0:6)
[forward demodulation 2018,316]
316. $sum(X0,0) = X0 (0:5) [theory axiom]
2018. ~$lesseq(c,0) | ~$lesseq(1,$sum(c,0)) (0:8)
[forward demodulation 1374,1613]
1613. c = a (0:3) [forward demodulation 1612,316]
1612. a = $sum(c,0) (0:5) [subsumption resolution 1611,1580]
1580. $lesseq(c,a) (0:3) [evaluation 1065]
1065. ~$lesseq(0,0) | $lesseq(c,a) (0:6)
[definition unfolding 751,1053,1054]
1054. a0 = 0 (0:3) [cnf transformation 307]
307. a0 = 0 [input prop4]
1053. c0 = 0 (0:3) [cnf transformation 306]
306. c0 = 0 [input prop2]
751. ~$lesseq(c0,a0) | $lesseq(c,a) (0:6) [cnf transformation 330]
330. $lesseq(c,a) | ~$lesseq(c0,a0) [flattening 5]
5. $lesseq(c,a) | ~$lesseq(c0,a0) [input inv4]
```

```
1611. ~$lesseq(c,a) | a = $sum(c,0) (0:8)
  [forward demodulation 1545,316]
1545. ~$lesseq($sum(c,0),a) | a = $sum(c,0) (0:10)
  [evaluation 1115]
1115. ~$lesseq(0,0) | ~$lesseq($sum(c,$uminus(0)),a) |
a = $sum(c,$uminus(0))(0:15)
  [definition unfolding 801,1054,1053,1053]
801. ~$lesseq(a0,0) | ~$lesseq($sum(c,$uminus(c0)),a) |
a = $sum(c,$uminus(c0)) (0:15) [cnf transformation 381]
381. a = $sum(c,$uminus(c0)) | ~$lesseq($sum(c,$uminus(c0)),a) |
~$lesseq(a0,0) [flattening 55]
55. a = $sum(c,$uminus(c0)) | ~$lesseq($sum(c,$uminus(c0)),a) |
~$lesseq(a0,0) [input inv54]
1374. ~$lesseq(a,0) | ~$lesseq(1,$sum(c,0)) (0:8) [evaluation 1350]
1350. ~$lesseq(a,0) | ~$lesseq(1,$sum(c,$uminus(0))) (0:9)
[definition unfolding 1036,1054,1053]
1036. ~$lesseq(a,a0) | ~$lesseq(1,$sum(c,$uminus(c0))) (0:9)
[cnf transformation 720]
720. ~$lesseq(1,$sum(c,$uminus(c0))) | ~$lesseq(a,a0)
  [flattening 290]
290. ~$lesseq(1,$sum(c,$uminus(c0))) | ~$lesseq(a,a0)
  [input inv289]
2061. $lesseq(c,0) (1:3) [resolution 1852,1057]
1057. $lesseq(0,X1) (0:3) [cnf transformation 746]
1852. ~$lesseq(0,$uminus(c)) | $lesseq(c,0) (0:7)
[forward demodulation 1256,1732]
1732. $sum(0,X0) = X0 (0:5) [backward demodulation 1731,1648]
1648. $sum(0,X0) = $sum(c,$sum(0,$sum($uminus(c),X0))) (0:12)
[forward demodulation 1129,1613]
1129. $sum(0,X0) = $sum(a,$sum(0,$sum($uminus(c),X0))) (0:12)
[definition unfolding 815,1054,1053]
815. $sum(a0,X0) = $sum(a,$sum(c0,$sum($uminus(c),X0))) (0:12)
[cnf transformation 396]
396. ! [X0] : $sum(a0,X0) = $sum(a,$sum(c0,$sum($uminus(c),X0)))
[rectify 69]
69. ! [X1] : $sum(a0,X1) = $sum(a,$sum(c0,$sum($uminus(c),X1)))
[input inv68]
1731. $sum(c,$sum(0,$sum($uminus(c),X0))) = X0 (0:10)
[forward demodulation 1730,316]
1730. $sum(X0,0) = $sum(c,$sum(0,$sum($uminus(c),X0))) (0:12)
[forward demodulation 1195,1613]
1195. $sum(X0,0) = $sum(a,$sum(0,$sum($uminus(c),X0))) (0:12)
[definition unfolding 881,1054,1053]
881. $sum(X0,a0) = $sum(a,$sum(c0,$sum($uminus(c),X0))) (0:12
```

```
) [cnf transformation 501]
501. ! [X0] : $sum(X0,a0) = $sum(a,$sum(c0,$sum($uminus(c),X0)))
[rectify 135]
135. ! [X28] : $sum(X28,a0) = $sum(a,$sum(c0,$sum($uminus(c),X28)))
[input inv134]
1256. $lesseq(c,0) | ~$lesseq(0,$sum(0,$uminus(c))) (0:9)
[definition unfolding 942,1053,1053]
942. $lesseq(c,0) | ~$lesseq(c0,$sum(c0,$uminus(c))) (0:9)
[cnf transformation 604]
604. ~$lesseq(c0,$sum(c0,$uminus(c))) | $lesseq(c,0)
  [flattening 196]
196. ~$lesseq(c0,$sum(c0,$uminus(c))) | $lesseq(c,0)
  [input inv195]
```

Starting from inv54 inferred by *Lingva* stating that either a is equal to $c - c0$ or c is not less or equal to a or the initial value of a is greater than 0 and with the two properties enforcing the initial values of the iterators *Vampire* infers formula (1611) stating that either c is grater than a or a equals c. Due to the fact that a is at least equal to c, property enforced by formula (1580). From the input inv289 expressing the fact that either c is greater than $c0$ or a is greater than $a0$ and the property stating the equality between a and c, *Vampire* infers that either c is equal to 0 or is greater or equal to 1(2019). From this formula and the negation of the invariant we are trying to prove $0 \leq c$ (2048) is inferred. From input invariants inv195 and inv68 *Vampire* infers that c is less or equal to 0. Applying resolution on the two formulas we obtain a refutation.

The Sum of Pairs Example. The next loop we analyze is "Sum of pairs".

```
 int m;
  int *aa;
  int x=getX(),
  l=0, u=m-1;

  //:$SORTED: A @ASC
while (l < u) {
 if (aa[l] + aa[u] < x)
      l = l+1;
   else
     if (aa[l] + aa[u] > x)
        u = u-1;
```

```
      else break;
%     }
```

This loop is particularly difficult to analyze with this method because it is constructed with a nested $if \dots then \dots else$ statement, which makes it difficult for *Lingva* to extract properties - as specified in the previous chapter, and the invariant we want to infer is inductive, type of property that can not be inferred straight forward by $Vampire$. For this specific loop we introduce the invariants inferred by $Cpp - inv$ and check if the theorem prover can deduce the invariant:

Property. 11 $aa[l] + aa[u] - x + 1 \leq 0$

The proof obtained by $Vampire$ is shown as follows:

```
Refutation found. Thanks to Tanya!
1898. $false (2:0) [subsumption resolution 1897,355]
355. $lesseq(0,l) (2:3) [resolution 208,103]
103. $lesseq(0,sK0) (0:3) [cnf transformation 80]
80. $lesseq(0,sK0) & $lesseq(sK0,l) &
~$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0)
[skolemisation 79]
79. ? [X0] : ($lesseq(0,X0) & $lesseq(X0,l) &
~$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0))
[flattening 78]
78. ? [X0] : (($lesseq(0,X0) & $lesseq(X0,l)) &
~$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0))
[ennf transformation 59]
59. ~! [X0] : (($lesseq(0,X0) & $lesseq(X0,l)) =>
$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0))
[rectify 24]
24. ~! [X3] : (($lesseq(0,X3) & $lesseq(X3,l)) =>
$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0))
[negated conjecture 23]
23. ! [X3] : (($lesseq(0,X3) & $lesseq(X3,l)) =>
$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0))
[input implication]
208. ~$lesseq(X16,sK0) | $lesseq(X16,l) (1:6) [resolution 31,104]
104. $lesseq(sK0,l) (0:3) [cnf transformation 80]
31. ~$lesseq(X1,X2) | ~$lesseq(X0,X1) | $lesseq(X0,X2) (0:9)
[theory axiom]
1897. ~$lesseq(0,l) (2:3) [subsumption resolution 1889,30]
30. $lesseq(X0,X0) (0:3) [theory axiom]
1889. ~$lesseq(l,l) | ~$lesseq(0,l) (2:6) [resolution 390,309]
309. ~$lesseq($sum(1,$sum($uminus(x),$sum(aa(u),aa(l)))),0) (0:12)
```

```
[forward demodulation 308,25]
25. $sum(X0,X1) = $sum(X1,X0) (0:7) [theory axiom]
308. ~$lesseq($sum(1,$sum($uminus(x),$sum(aa(l),aa(u)))),0) (0:12)
[forward demodulation 307,25]
307. ~$lesseq($sum(1,$sum($sum(aa(l),aa(u)),$uminus(x))),0) (0:12)
[forward demodulation 306,279]
279. $sum(X7,$sum(X8,X9)) = $sum(X9,$sum(X7,X8)) (1:11)
[superposition 26,25]
26. $sum(X0,$sum(X1,X2)) = $sum($sum(X0,X1),X2) (0:11)
[theory axiom]
306. ~$lesseq($sum($uminus(x),$sum(1,$sum(aa(l),aa(u)))),0) (0:12)
[forward demodulation 305,279]
305. ~$lesseq($sum($uminus(x),$sum(aa(l),$sum(aa(u),1))),0) (0:12)
[forward demodulation 304,25]
304. ~$lesseq($sum($uminus(x),$sum($sum(aa(u),1),aa(l))),0) (0:12)
[forward demodulation 296,279]
296. ~$lesseq($sum(aa(l),$sum($uminus(x),$sum(aa(u),1))),0) (0:12)
[backward demodulation 279,143]
143. ~$lesseq($sum(aa(l),$sum(aa(u),$sum(1,$uminus(x)))),0) (0:12)
[forward demodulation 105,25]
105. ~$lesseq($sum(aa(l),$sum(aa(u),$sum($uminus(x),1))),0) (0:12)
[cnf transformation 80]
390. $lesseq($sum(1,$sum($uminus(x),$sum(aa(u),aa(X0)))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (1:18) [superposition 314,25]
314. $lesseq($sum(1,$sum($uminus(x),$sum(aa(X0),aa(u)))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 313,25]
313. $lesseq($sum(1,$sum($sum(aa(X0),aa(u)),$uminus(x))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 312,279]
312. $lesseq($sum($uminus(x),$sum(1,$sum(aa(X0),aa(u)))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 311,279]
311. $lesseq($sum($uminus(x),$sum(aa(u),$sum(1,aa(X0)))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 310,26]
310. $lesseq($sum($uminus(x),$sum($sum(aa(u),1),aa(X0))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 297,279]
297. $lesseq($sum(aa(X0),$sum($uminus(x),$sum(aa(u),1))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [backward demodulation 279,140]
140. $lesseq($sum(aa(X0),$sum(aa(u),$sum(1,$uminus(x)))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [forward demodulation 100,25]
100. $lesseq($sum(aa(X0),$sum(aa(u),$sum($uminus(x),1))),0) |
~$lesseq(X0,l) | ~$lesseq(0,X0) (0:18) [cnf transformation 73]
73. ! [X0] : (~$lesseq(0,X0) | ~$lesseq(X0,l) |
$lesseq($sum(aa(X0),$sum(aa(u),$sum($uminus(x),1))),0))
  [flattening 72]
72. ! [X0] : ((~$lesseq(0,X0) | ~$lesseq(X0,l)) |
```

```
$lesseq($sum(aa(X0),$sum(aa(u),$sum($uminus(x),1))),0))
[ennf transformation 56]
56. ! [X0] : (($lesseq(0,X0) & $lesseq(X0,l)) =>
$lesseq($sum(aa(X0),$sum(aa(u),$sum($uminus(x),1))),0))
  [rectify 20]
20. ! [X3] : (($lesseq(0,X3) & $lesseq(X3,l)) =>
$lesseq($sum(aa(X3),$sum(aa(u),$sum($uminus(x),1))),0))
  [input pro3]
```

From the input property that represents one of the invariants discovered by $Cpp-inv$, $\forall X3((0 \leq X3 \wedge X3 \leq l \Rightarrow aa(X3)+aa(u)+1-x \leq 0))$ using forward demodulation in combination with theory axioms $Vampire$ infers that $\$lesseq(0,l)$. But in the negated conjecture we have the negation of this clause resulting into a refutation.

This result is not surprising, taking into account that $Cpp-inv$ can infer most of the invariants that are inductive so the formulas added to *Lingva*s result cover the *Vampire*s lack using induction. The great number of theories that are implemented in the theorem prover makes the inference of the invariant possible, the formula introduced as an axiom being processed and modeled by this theorems to take the form we were looking for.

The Sequential Initialization Example. Also in the case of the loop for the program "Sequential initialization " proving the property of interest is not straight forward.

```
 int main()
{
  int m;
  int *aa;

  aa[0]=7;
  int i=1;
  while (i<m)
  {
    aa[i]=aa[i-1]+1;
    ++i;
  }
}
```

Since in the program there are specified initial values for the iterator i and for the first element in the array aa but *Lingva* did not infer them automatically we introduce them by hand in the form of two properties:

Property. 12 *tff(prop1, axiom, i0=1).*

and

Property. 13 *tff(prop3, axiom, aa0(0)=7).*

We observe that *Lingva* managed to infer some very useful invariants stating that the value of an element in *aa* that is in a position beyond the boundaries (less than 0, or greater than i, greater than m) the value of the element does not change. We introduce a property expressing the fact that if the element lies between the boundaries of the processed array its value is changed:

Property. 14 *tff(propx, axiom, ![X:$int]:*
(($lesseq($sum(i,$uminus(i0)),sK0(X)) & $sum(i0,sK0(X)) = X
& $lesseq(0,sK0(X)) & $lesseq(m,$sum(i0,sK0(X))))=> ¬aa(X)=aa0(X)
)).

Since there is no property inferred about the way a value is changed by this loop (although we already have this information from the output of Cpp-inv) we introduce a new formula stating this relation:

Property. 15 *tff(prop10, axiom, ![X:$int]: (aa(X)=aa0(X) |*
aa(X)=$sum(aa($sum(X,$uminus(1))),1))).

With these new formulae added *Vampire* can prove that: $\forall X, (0 \leq X) \wedge (X < i) \Rightarrow aa(X) == aa(X-1)+1$. The proof outputted is give as follows:

```
Refutation found. Thanks to Tanya!
5511. $false (0:0) [subsumption resolution 5508,702]
702. sP2(sK0) (0:2) [inequality splitting 697,701]
701. ~sP2($sum(1,sK0(sK0))) (0:5)
[inequality splitting name introduction]
697. $sum(1,sK0(sK0)) != sK0 (0:6) [definition unfolding 544,539]
539. i0 = 1 (0:3) [cnf transformation 151]
151. i0 = 1 [input prop1]
544. $sum(i0,sK0(sK0)) != sK0 (0:6) [cnf transformation 388]
388. $lesseq($sum(i,$uminus(i0)),sK0(sK0)) & $sum(i0,sK0(sK0))!=
sK0 & ~$lesseq(0,sK0(sK0)) & $lesseq(m,$sum(i0,sK0(sK0))) &
aa(sK0) != $sum(aa($sum(sK0,$uminus(1))),1) [skolemisation 387]
155. ! [X37] : (($lesseq($sum(i,$uminus(i0)),sK0(X37)) &
~$sum(i0,sK0(X37)) = X37 & ~$lesseq(0,sK0(X37)) &
$lesseq(m,$sum(i0,sK0(X37)))) =>
aa(X37) = $sum(aa($sum(X37,$uminus(1))),1))
[input implication]
5508. ~sP2(sK0) (0:2) [backward demodulation 5504,701]
5504. $sum(1,sK0(sK0)) = sK0 (1:6)
```

```
[subsumption resolution 5503,545]
545. ~$lesseq(0,sK0(sK0)) (0:4) [cnf transformation 388]
5503. $lesseq(0,sK0(sK0)) | $sum(1,sK0(sK0)) = sK0 (1:10)
[subsumption resolution 5497,696]
696. $lesseq(m,$sum(1,sK0(sK0))) (0:6)
[definition unfolding 546,539]
546. $lesseq(m,$sum(i0,sK0(sK0))) (0:6) [cnf transformation 388]
5497. ~$lesseq(m,$sum(1,sK0(sK0))) | $lesseq(0,sK0(sK0)) |
$sum(1,sK0(sK0)) = sK0 (1:16) [resolution 909,912]
912. $lesseq($sum(-1,i),sK0(sK0)) (0:6)
[forward demodulation 703,157]
157. $sum(X0,X1) = $sum(X1,X0) (0:7) [theory axiom]
703. $lesseq($sum(i,-1),sK0(sK0)) (0:6) [evaluation 698]
698. $lesseq($sum(i,$uminus(1)),sK0(sK0)) (0:7)
[definition unfolding 543,539]
543. $lesseq($sum(i,$uminus(i0)),sK0(sK0)) (0:7)
[cnf transformation 388]
909. ~$lesseq($sum(-1,i),sK0(X0)) | ~$lesseq(m,$sum(1,sK0(X0))) |
$lesseq(0,sK0(X0)) | $sum(1,sK0(X0)) = X0 (0:22)
[forward demodulation 908,157]
908. ~$lesseq(m,$sum(1,sK0(X0))) | $lesseq(0,sK0(X0)) |
$sum(1,sK0(X0)) = X0 | ~$lesseq($sum(i,-1),sK0(X0)) (0:22)
[subsumption resolution 705,548]
548. ~$lesseq(m,$sum(1,sK0(X0))) | aa(X0) = aa0(X0) (0:11)
[definition unfolding 389,539]
389. ~$lesseq(m,$sum(i0,sK0(X0))) | aa(X0) = aa0(X0) (0:11)
[cnf transformation 170]
170. ! [X0] : (aa(X0) = aa0(X0) | ~$lesseq(m,$sum(i0,sK0(X0))))
[flattening 1]
1. ! [X0] : (aa(X0) = aa0(X0) | ~$lesseq(m,$sum(i0,sK0(X0))))
[input inv0]
705. aa(X0) != aa0(X0) | ~$lesseq(m,$sum(1,sK0(X0))) |
$lesseq(0,sK0(X0)) | $sum(1,sK0(X0)) = X0 |
~$lesseq($sum(i,-1),sK0(X0)) (0:27) [evaluation 695]
154. ! [X37] : (($lesseq($sum(i,$uminus(i0)),sK0(X37)) &
~$sum(i0,sK0(X37)) = X37 & ~$lesseq(0,sK0(X37)) &
$lesseq(m,$sum(i0,sK0(X37)))) => ~aa(X37) = aa0(X37))
[input propx]
```

Vampire infers from the input negated implication, from the invariant *inv*0 deduced by *Lingva*, stating the connection between the relation between the values of i and m and not modifying the values in array aa,

property (5504) $\$sum(1, sK0(sK0)) = sK0$. From this formula and $\neg sP2(\$sum(1, sK0(sK0)))$ (701), which is an inequality splitting name introduction, formula $sP2(sK0)$(5508) is inferred. From this formula and the negated invariant *Vampire* obtains a refutation.

The Insertion Example. The loop *Insertion* requires an invariant that is inductive so *Vampire* can not prove this invariant without some extra knowledge added to the invariants inferred by *Lingva*.

```
 x=aa[i];
j = i-1;
  while (j >= 0 and
       aa[j] > x) do
    aa[j+1] = aa[j];
    --j;
  end do
```

We observe a few properties that were already discovered by *Lingva* : there are four properties that express the fact that if the position in the array is out of the bounds (smaller than 0, greater than j) the value of the elements is not modified. There is also a property communicating the fact that if j is greater than 0, than for the element at the initial value of j (at $j0$) the property that the value is shifted one to the right holds. Based on this formulas we take the decision of introducing the following property stating that all elements that are at a position higher than j are greater than the element at this position.

Property. 16 *tff(prop4, axiom, ![X:$int]:($lesseq(j,X)| $lesseq(X,j0)| $less(aa(j),aa(X))))*.

The proof found by *Vampire* for the invariant is reproduced in the following lines:

```
 Refutation found. Thanks to Tanya!
1299. $false (1:0) [subsumption resolution 1298,160]
160. $lesseq(X0,X0) (0:3) [theory axiom]
1298. ~$lesseq(j,j) (1:3) [subsumption resolution 1295,856]
856. ~$lesseq(X0,j) | $lesseq(X0,j0) (3:6) [resolution 839,161]
161. ~$lesseq(X1,X2) | ~$lesseq(X0,X1) | $lesseq(X0,X2) (0:9)
[theory axiom]
839. $lesseq(j,j0) (2:3) [resolution 809,639]
639. $lesseq(j,sK0) (1:3) [resolution 162,530]
530. ~$lesseq(sK0,j) (0:3) [cnf transformation 379]
379. ~$lesseq(sK0,j) & $lesseq(sK0,j0) & $lesseq(aa(sK0),x)
[skolemisation 378]
```

```
378. ? [X0] : (~$lesseq(X0,j) & $lesseq(X0,j0) & $lesseq(aa(X0),x))
[ennf transformation 377]
377. ~! [X0] : ($lesseq(X0,j) | ~$lesseq(X0,j0) |
~$lesseq(aa(X0),x)) [flattening 376]
376. ~! [X0] : (~~$lesseq(X0,j) | ~$lesseq(X0,j0) |
 ~$lesseq(aa(X0),x))[rectify 154]
154. ~! [X36] : (~~$lesseq(X36,j) | ~$lesseq(X36,j0) |
~$lesseq(aa(X36),x)) [evaluation 152]
152. ~! [X36] : (~$less(j,X36) | ~$lesseq(X36,j0) |
~$lesseq(aa(X36),x))[negated conjecture 151]
151. ! [X36] : (~$less(j,X36) | ~$lesseq(X36,j0) |
~$lesseq(aa(X36),x))[input implication]
162. $lesseq(X0,X1) | $lesseq(X1,X0) (0:6) [theory axiom]
809. ~$lesseq(X24,sK0) | $lesseq(X24,j0) (1:6) [resolution 161,531]
531. $lesseq(sK0,j0) (0:3) [cnf transformation 379]
1295. ~$lesseq(j,j0) | ~$lesseq(j,j) (1:6) [resolution 529,160]
529. ~$lesseq(aa(X0),aa(j)) | ~$lesseq(X0,j0) | ~$lesseq(j,X0)
(0:11)[cnf transformation 375]
375. ! [X0] : (~$lesseq(j,X0) | ~$lesseq(X0,j0) |
~$lesseq(aa(X0),aa(j)))[flattening 374]
374. ! [X0] : (~$lesseq(j,X0) | ~$lesseq(X0,j0) |
~$lesseq(aa(X0),aa(j)))[rectify 153]
153. ! [X36] : (~$lesseq(j,X36) | ~$lesseq(X36,j0) |
~$lesseq(aa(X36),aa(j)))[evaluation 150]
150. ! [X36] : (~$lesseq(j,X36) | ~$lesseq(X36,j0) |
$less(aa(j),aa(X36)))[input prop4]
```

The proof for this invariant is less complex than the other proofs. From the formula 839 stating that $j \leq j0$ *Vampire* infers that X is either grater than j or smaller than $j0$(856). From this and negated invariant $j > j$ is inferred which is in contradiction with the property introduced above.

5.2 Discussions and Conclusions

We seen that every one of the studied tools has its own advantage regarding the strength or type of inferred invariant. We observe that we get significantly better results when we use the invariant inferred by one tool as input for the others in order to get even stronger invariants. One remark is in order at this step and that is that the form of the formula that we try to infer with *Vampire* is important with respect to the set of formulas that are provided as input. Although using human intuition the invariant may be

found just as hard to infer regardless of the form that it has, for the theorem prover a decision must be made on how many times should the basic theorems (such as commutativity, associativity) be applied on a formula and in which order. An important step in this field is the automatization of the inference process and the combination of techniques.

6 Conclusions

We have provide an extensive evaluation of the state of the art in invariant generation techniques. Although the techniques are not mutually exclusive, in the sens that the same invariant can be generated by more than one technique, we found a pattern of invariants that can or can't be generated by a certain method. In order to infer certain properties about programs, this classification can point to the user the method that is most likely to infer it.

The set of programs for which we studied the behavior of these tools were chosen from a series of loops on which either one of the tools had difficulties analyzing, the loops were representative for the type of invariants that a certain tool could infer or the invariants of interest for the loop had an interesting structure. Although this set is not large, the invariants that were inferred cover a series of patterns that occur often in program verification.

We further studied the disadvantages of *Lingva* and *Vampire* and improve their functionality by either combining the result of *Lingva* with results of other methods or by adding properties that are easy to observe by the user. We choose the set of properties to add for each loop depending on its structure and also on the known functionality of the tool. To this end we managed to infer invariants of interest for every loop studied only with the cost of writing theorems hard to infer by this tool.

Invariants of interest for the user are hard to fined since there might be a large number of properties between two or more variables from the program. Nevertheless the three tools managed to infer invariants that would help the user understand the program better. Also the saturation theorem prover and post-condition weakening methods are goal oriented since both require input from the user, having an advantage of inferring the invariant the user is interested in.

Due to program verification undecidability the task of inferring invariants is hard. Selecting a suitable benchmark for an evaluation of techniques developed for this task is not trivial since the programs come in a large variety of languages and combination of statements. We seen in our evaluation that combining such techniques have a positive influence on the result. This observation take as to the conjecture that a promising path for future work is to combine methods that have different advantages and evaluate them on loops that have a more complicated behavior.

We also observed that there is significant amount of cases in which human interfering with the procedure (such as adding theorems or stating postconditions) provides a better result from the users point of view. This information provides the intuition that combining human intuition can play an important part in solving the invariant generation task. Algorithms used

in machine learning, that simulate human rationality can be an asset for this section of program verification. To the best of our knowledge this concept was not used yet so for future work this is an interesting path that could be followed.

7 References

[AAR09] Cimatti A., Griggio A., and Sebastiani R. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. In *ACM Transactions on Computational Logic* , volume 12, October 2009.

[And02] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. 2002.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking . 58, 2003.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs . *FMCO*, 4111:364–387, 2005.

[BMSW10] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie—An Interactive Prover-Backend for the Verifying C Compiler. 44:111–144, February 2010.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints . pages 238–252, 1977.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. pages 151–158, 1971.

[DdM06] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T) . pages 81–94, 2006.

[Dij75] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. 18:453–457, 1975.

[DL62] Martin Davis, George Logemann , and Donald Loveland. A machine program for theorem-proving. 5:394–397, July 1962.

[dMB11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. 54:69–77, 2011.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. 7:201–215, July 1960.

[DW50] Hilbert D. and Ackermann W. *Principles of Mathematical Logic*. Chelsea Publishing Company, 1950.

[ES04] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. 2919:502–518, 2004.

[FM10] Carlo A. Furia and Bertrand Meyer. Inferring Loop Invariants using Postconditions . In *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*. 2010.

[G.S83] G.S.Tseitin. On the Complexity of Derivation in Propositional Calculus. pages 466–483, 1983.

[HHKR10] Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrey Rybalchenko. Aligators for Arrays (Tool Paper) . 6397: 348–356, 2010.

[HKV11] Krystof Hoder, Laura Kovacs, and Andrei Voronkov. Case Studies on Invariant Generation Using a Saturation Theorem Prover. In *10th Mexican International Conference on Artificial Intelligence, MICAI*, 2011.

[KV09] Laura Kovacs and Andrei Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*, 2009.

[KVar] Laura Kovacs and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of the International Conference on Computer Aided Veri cation (CAV), LNCS*, 2013 to appear.

[LRCR13] Daniel Larraz, Enric Rodriguez-Carbonell, and Albert Rubio. SMT-Based Array Invariant Generation. In *14th International Conference Verification, Model Checking, and Abstract Interpretation, VMCAI*, 2013.

[MMZ+11] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver . 2011.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability . pages 506–521, May 1999.

[SS09] Strivastava S. and Gulwani S. Program Verification using Template over Predicate Abstraction. In *Proc. of PLDI*, 2009.

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. 4130:292–297, 2006.

[Wan95] Jinchang Wang. A branching heuristic for testing propositional satisfiability . 5, October 1995.