Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master in Computing

Master of Science Thesis

Automatic generation of loop invariants

Daniel Larraz Hurtado

Advisor: Albert Rubio Gimeno

June, 2011

Contents

1 Overview

In this thesis we present CppInv, an automatic loop invariants generator for imperative programs written in a subset of the C++ language. Firstly, we start with an introduction to the problem and the related work. After that, we give a general perspective of the tool design. Before concluding this chapter, we describe the organization of the rest of the document.

1.1. Introduction

An invariant assertion of a program at a location is an assertion over the program variables that remains true whenever the location is reached. Discovering invariants is an essential task for verifying the correctness of programs. Since reliable software design and implementation continues to be an important problem, any progress in this area will be extremely important in the future.

The field of invariant generation is based on a multitude of techniques such as computer algebra, theorem proving, constraint solving, abstract interpretation techniques or model-checking.

A particularly interesting case is the generation of loop invariants for imperative programs or transition systems in general. In this context, abstract interpretation [2] is the classical technique for invariant generation. The main idea behind this approach is to perform an approximate symbolic execution of the program until an assertion that remains unchanged is reached. However, in order to guarentee termination, the method introduces imprecision by use of an extrapolation operator called *widening*. This operator often causes the technique to produce weak invariants. The design of a widening operator with some guarantee of completeness remains a key challenge for abstract interpretation based techniques [12,13].

In this thesis, we will consider an alternative method for automatically generate linear invariants, i.e. invariants expressed as a linear inequation, which was presented in [1]. This method uses Farkas Lemma to transform the problem of the existence of a loop invariant into a satisfiability problem in propositional logic over non-linear integer arithmetic. Despite the potential of the method, its application [14] has been limited due to the lack of good solvers for the obtained non-linear constraints.

Solving non-linear arithmetic constraint over the integers is undecidable. The situation is not much better when considering the reals since, although the problem is decidable as it was shown in [4], using the related algorithms in practice is unfeasible due to their complexity.

Therefore, all methods used in practice for both integer or real solution domains are incomplete and are focused on either proving satisfiability or proving unsatisfiability. In this thesis we are particularly interested on the former because each found solution represents a new discovered invariant. That is the reason we choose the new solver called Barcelogic presented in [3] to discover the invariants.

1.2. Tool design

CppInv works in two stages. Firstly, it parses a source code written in a subset of C++ and abstracts all execution paths of the program building a control flow graph associated to a transition system. Paths are expressed as arbitrary propositional formulas over linear integer arithmetic including high level operators like integer division and modulo. That makes easy the initial modeling. Later, formulas are normalized and only paths between a set of locations that cover every cycle of the control flow graph are regarded.

Secondly, CppInv generates linear invariants at the selected locations setting out a constraint solving problem. We present a method to discover all linear invariant of the considered form.

As a result, our tool can find linear invariants efficiently for a large set of interesting programs. Moreover, CppInv is also able to generate some non-linear invariants automatically. For instance, it is possible to prove the total correctness of a program that multiplies two integers from the invariants returned by the tool.

1.3. Structure of the document

The rest of the document is organized as follows: Chapter 2 introduces some basic concepts and definitions. Chapter 3 describes the input language accepted by the tool and the abstraction used to model an input program. Chapter 4 presents a method for generating a set of linear invariants for each loop point of a program. Chapter 5 explains some aspects of the tool implementation. Chapter 6 presents the results of this thesis and futher work. Appendix A collects examples taken from several papers and discusses the outcome obtained using the tool.

2 Preliminaries

For reading this document, some knowledge of propositional and first-order logic is assumed (see [9]), although the notation and the main concepts used in this work are recalled in this chapter. Besides, we introduce here the notion of SMT problems, integer linear formulas, Presburger formulas and transition systems, which will be necessary to follow next chapters.

2.1. SMT formulas and SMT problems

Definition 1 (SMT Formulas) Let P be a fixed finite set of *ground* (i.e., variable free) first-order atoms. The set of *Satisfiability Modulo Theories (SMT) formulas* over P is the set of boolean formulas over P defined as follows:

- Every $p \in P$ is an SMT formula.
- If F and G are SMT formulas, $\neg F$, $(F \wedge G)$ and $(F \vee G)$ are also SMT formulas.
- Nothing else is an SMT formula.

If $p \in P$, then p is an *atom* and p and $\neg p$ are *literals* of P. We will write $(F \rightarrow G)$ as an abbreviation for $(\neg F \vee G)$.

A formula is in *Negation Normal Form (NNF)* if negation occurs only immediately above an atom, and, it is in *Disjunctive Normal Form (DNF)* if it is a disjunction of conjunctions of literals.

Recall that given an arbitrary formula F, we can obtain its NNF applying the next three transformation rules up to completion:

$$F[\neg\neg G] \Longrightarrow F[G]$$
$$F[\neg(G \wedge H)] \Longrightarrow F[\neg G \vee \neg H]$$
$$F[\neg(G \vee H)] \Longrightarrow F[\neg G \wedge \neg H]$$

Moreover, once a formula F is in NNF, we can obtain its DNF applying the distributivity rule up to completion:

$$F[I \wedge (G \vee H)] \Longrightarrow F[(I \wedge G) \vee (I \wedge H)]$$

We will use SMT formulas with theories. A *theory* T is a set of closed first-order formulas. An (SMT) formula F is *T-satisfiable* if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called *T-unsatisfiable*. We will say M is a *T-model* of a formula F over P, written $M \models F$, if M is an interpretation over P that T-safisfies F. Two formulas F and G are *logically equivalent*, denoted $F \equiv G$, if F and G have the same T-models. A formula G is a a *logical consequence* of a formula F, denoted $F \models G$, if every T-model of F is a T-model of G. Recall entailment can be captured with the following known lemma.

Lemma 1. Given a T-model M and two formulas F and G, $M \models (F \rightarrow G)$ if and only if $(M \models F)$ implies $(M \models G)$, i.e, G is a logical consequence of F.

In order to generate program invariants we will set out and solve SMT problems. An *SMT problem* for a theory T is the problem of determining, given an SMT formula F, whether F is T-satisfiable, or, equivalently, whether F has a T-model.

When a theory T is implied by the context we will omit prefixes, for instance, writing satisfiable instead of T-satisfiable.

2.2. Integer Linear formulas and Presburger formulas

Definition 2 (Integer Linear Formula) An *integer linear term* is either c, cx, $c_1\frac{E}{c_2}$ (division), or $c_1(E\%c2)$ (modulus), for constants $c, c_1 \in \mathbb{Z}$, positive constant $c_2 \in \mathbb{Z}^+$, integer variable x, and integer linear expression E. An *integer linear expression* is the summation of integer linear terms. An *integer linear atom* is the comparison $E_1 \bowtie E2$ of two integer linear expressions, for $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$. An *integer linear formula* is an SMT formula over integer linear atoms.

A integer linear formula F is in *Integer Lineal Negation Form* (ILNF) if it is in negation normal form and every integer linear atom $E_1 \bowtie E2$ contained in F is negated applying the next standard rules:

$$(E_1 = E_2) \Longrightarrow (E_1 \neq E_2), (E_1 \neq E_2) \Longrightarrow (E_1 = E_2)$$
$$(E_1 < E_2) \Longrightarrow (E_1 \geq E_2), (E_1 \leq E_2) \Longrightarrow (E_1 > E_2)$$
$$(E_1 > E_2) \Longrightarrow (E_1 \leq E_2), (E_1 \geq E_2) \Longrightarrow (E_1 < E_2)$$

Definition 3 (Presburger Formula) A *Presburger formula* is an integer linear formula that does not involve division or modulo arithmetic. Given an integer linear formula F, an equisatisfiable Presburger formula is obtained by repeatedly applying the next rules (taken from [5]):

$$F[c_1\frac{E}{c_2}]) \Longrightarrow \bigvee_{b\in[0..c_2-1]} (\exists a)\begin{pmatrix} [c_2a + b = E \wedge E \geq 0 \wedge F[c_1a]] \\ \vee\, [c_2a - b = E \wedge E \leq 0 \wedge F[c_1a]] \end{pmatrix}$$

$$F[c_1(E\%c_2)]) \Longrightarrow \bigvee_{b\in[0..c_2-1]} (\exists a)\begin{pmatrix} [c_2a + b = E \wedge E \geq 0 \wedge F[c_1b]] \\ \vee\, [c_2a - b = E \wedge E \leq 0 \wedge F[-c_1b]] \end{pmatrix}$$

A Presburger formula is in *Equality Normal Form* (ENF) if it does not contain inequalities. Given a Presburger formula F, we can obtain its ENF applying the next transformation rule up to completion:

$$F[E_1 \neq E_2] \Longrightarrow F[(E_1 < E_2) \vee (E_1 > E_2)]$$

In next chapter, we will use integer linear formulas for modeling assertions over program variables as close to the input program instructions as possible. Then, we will convert them to normalized Presburger formulas which will allow us to set out the SMT problem whose solutions give us the loop invariants of the input program.

2.3. Transition Systems

Definition 4 (Transition System) A *transition system* P: $\langle V, L, l_0, \mathcal{T}\rangle$ consists of a set of *variables* V, a set of *locations* L, an *initial location* l_0, and a set of transitions $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a tuple $\langle l, l', \rho_\tau\rangle$, where $l, l' \in L$ are the *pre* and *post locations*, and ρ_τ is the *transition relation*, an assertion over $V \cup V'$, where V represents current-state variables and its primed version V' represents the next-state variables.

A transition system is an *integer linear transition system* (IL-TS) if its variables are integers and all of its transition relations are integer linear formulas.

A transition system is a *Presburger transition system* (P-TS) if its variables are integers and all of its transition relations are Presburger formulas.

Example 1. An example of Presburger transition system is shown below:

$L = \{l_0, l_1, l_2, l_3, l_4, l_5\}, V = \{x, y, z\}$

$\tau_0 = \langle l_0, l_1, x' = 3 \wedge y' = 13 \wedge z' = 7\rangle$

$\tau_1 = \langle l_1, l_2, (x < z)\rangle$

$\tau_2 = \langle l_2, l_4, x' = x + 1\rangle$

$\tau_3 = \langle l_1, l_3, (z \leq x)\rangle$

$\tau_4 = \langle l_3, l_4, z' = z + 1\rangle$

$\tau_5 = \langle l_4, l_1, (x < y \wedge z < y)\rangle$

$\tau_6 = \langle l_4, l_5, (y \leq x \vee z \geq y)\rangle$

Throughout the document, unless otherwise stated, we assume that the set of variables $V = \{x_1, \cdots, x_n\}$ is fixed. Furthermore, given an assertion ψ over the variables V of a transition system, ψ' denotes the assertion obtained by replacing each variable $x \in V$ by $x' \in V$.

The *control-flow graph* (CFG) of a transition system is a graph whose vertices are the locations and whose edges are the transitions. A path π of the transition system is a path through its CFG, and the relation ρ_π associated with the path is the composition of the corresponding transition relations.

A *cutset* C of a transition system P is a subset of the locations of P with the property that every cyclic path in P passes through some location in C. A location inside a cutset is called a *cutpoint*. A *basic* path π between two cutpoints l_i and l_j is a simple path that does not go through any cutpoint, other than the end points.

In the next chapter we will use transition systems to model imperative programs. In order to make easy the translation while being compliant with imperative programs' semantic, we introduce the following assumption. Given the relation ρ_π associated with a path π of an IL-TS or a P-TS with set of variables V and a variable $x \in V$, we assume that ρ_π contains an atom $x' = x$ unless ρ_π already includes an atom of the form $x' = E$, where E is an integer linear expression or a Presburger expression depending on the type of the transition system considered. This assumption ensures that a variable that is not assigned through an excution path keeps its previous value.

Example 2. All cycles in the P-TS of example 1 are cut by l_3. A path that cycle back to l_3 is π_1 using τ_1, τ_2 and τ_5. The relation ρ_{π_1} associated with π_1 is $(x < z) \wedge x' = x + 1 \wedge (x < y) \wedge (z < y) \wedge y' = y \wedge z = z'$ where $(y' = y \wedge z = z')$ has been assumed.

3 Input program abstraction

In this chapter we introduce the input language accepted by CppInv. Next, we describe how to model an input program with an integer linear transition system. Finally, we explain which transformations are necessary before we can set out the SMT problem that generates the loop invariants.

3.1. Input language

CppInv takes as input a program over integer and boolean expressions and assignments given in a subset of the C++ language. Specifically, it accepts a `main` function definition (`int main() {...}`) which can include declarations and initializations of integer variables (`int x=...`), sequential statements (`stmt1;...;stmtN`), `if-else` conditional statements, `while` and `do-while` loops, `break` and `continue` statements, label declarations and `goto` jumps, integer arithmetic expressions using addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`) and modulo (`%`) operators, boolean expressions using conjunctions (`and`, `&&`), disjunctions (`or`, `||`) and negations (`not`, `!`) of boolean values (`true`, `false`) and comparisons of integer expressions using equality, inequality and relational operators (`=`,`!=`,`<`, `<=`,`>`,`>=`), assignments (`=`, `+=`, `-=`, `*=`, `/=`, `%=`), pre and post increment (`++`) and decrement (`--`) statements and assertions of boolean expressions (`assert(...)`).

Additionally, CppInv has syntactic support for integer arrays and function calls and partial support for boolean variables. An input program can declare an integer array (`int A[...]`) and access and update (`A[...]=...`) its element values but, however, those actions are semantically ignored by the tool. A parsed function call (`name(...)`) is always treat as a call to an non-deterministic function that returns an arbitrary integer or a boolean value, depending on the context. Unlike integer variables, boolean variables are not represented directly in the used logic. Thus, no assertion about a boolean variable state is possible. However, boolean variables are replaced with the last expression assigned to them at those locations where they are used for having more information about integer variables. This technique, although is useful, is limited since it is not always possible to determine which has been the last assigned expression when the program reaches some program location.

Notice that according to the explained features of the input language, other programming languages could be chosen instead of C++. Nevertheless, C++ was chosen since high level features of the language are expected to be supported in future doctoral work.

Example 3. Next figure shows an accepted input program that multiplies two positive integers `m1` and `m2`[1].

[1] This example is based on one taken from [5]

```
int main() {
  // Pre: m1=M1 /\ m2=M2
  int m1, m2, p=0;
  assert(m1>=1 && m2>=1);
  while (m1>0) {
    if (m1%2==1) p=p+m2;
    m1=m1/2;
    m2=2*m2;
  }
  // Post: p = M1*M2
}
```

3.2. Program modeling as Transition System

The first task of CppInv is to parse the source code of a program given in C++ syntax and build an integer linear transition system that represents it. The translation is applied in the standard way. Only some cases are problematic and deserve a special metion. For instance, when an integer variable is assigned to a non-linear integer expression or the value returned by a function call, a new auxiliary variable that represents an unknown arbitrary value is used instead. Note that if we simply remove the instruction it would not be possible to reference this value if the variable is used later in the program.

Example 4. Next figure shows the integer linear transition system corresponding to Example 3 together with the labeled version of the program:

$L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7\}, V = \{m1, m2, p\}$

$\tau_0 = \langle l_0, l_1, p' = 0\rangle$

$\tau_1 = \langle l_1, l_2, (m1 \geq 1) \wedge (m2 \geq 1)\rangle$

$\tau_2 = \langle l_2, l_3, (m1 > 0)\rangle$

$\tau_3 = \langle l_3, l_4, (m1\%2 = 1)\rangle$

$\tau_4 = \langle l_4, l_5, p' = p + m2\rangle$

$\tau_5 = \langle l_3, l_5, \neg(m1\%2 = 1)\rangle$

$\tau_6 = \langle l_5, l_6, m1' = m1/2\rangle$

$\tau_7 = \langle l_6, l_2, m2' = 2 * m2\rangle$

$\tau_8 = \langle l_2, l_7, \neg(m1 > 1)\rangle$

```
int main()
{
    // Pre: m1=M1 /\ m2=M2
    int m1, m2;
l0: int p=0;
l1: assert(m1>=1 && m2>=1);
l2: while (m1>0) {
l3:   if (m1%2==1)
l4:     p=p+m2;
l5:   m1=m1/2;
l6:   m2=2*m2;
    }
l7: // Post: p = M1*M2
}
```

3.3. Transition System normalization

Once an integer linear transition system $\mathcal{P}$ is obtained from an input program, we start a sequence of normalizing steps:

1. Every transition relation in $\mathcal{P}$ is converted into its Negation Normal Form.
2. Integer linear transition system $\mathcal{P}$ is transformed into a Presburger transition system $\mathcal{P}'$.
3. Every transition relation in $\mathcal{P}'$ is changed into its Equality Normal Form.
4. Every transition relation in $\mathcal{P}'$ is converted into its Disjunctive Normal Form.
5. Every transition relation of the form $C_1 \vee \ldots, \vee C_n$ with $n > 1$ is replaced with n transition relation, one for ervery C_i.

The aim of steps 1 and 2 is to remove division operators which can not be handled by the SMT solver used, while the goal of steps 3, 4 and 5 is to make explicit a set of transitions encoded implicitly as disjunctions in the logic.

Example 5. Next figure shows the Presburger transition system obtained after applying step 1 and 2 to the running example.

$$L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7\}, V = \{m1, m2, p, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$$

$$\tau_0 = \langle l_0, l_1, p' = 0\rangle$$

$$\tau_1 = \langle l_1, l_2, (m1 \geq 1) \wedge (m2 \geq 1)\rangle$$

$$\tau_2 = \langle l_2, l_3, (m1 > 0)\rangle$$

$$\begin{aligned}\tau_3 = \langle l_3, l_4, &(2 * \beta_1 = m1 \wedge m1 \geq 0 \wedge 0 = 1) \vee (2 * \beta_1 = m1 \wedge m1 \leq 0 \wedge 0 = 1) \vee \\ &(2 * \beta_2 + 1 = m1 \wedge m1 \geq 0 \wedge 1 = 1) \vee (2 * \beta_2 - 1 = m1 \wedge m1 \leq 0 \wedge -1 = 1)\rangle\end{aligned}$$

$$\tau_4 = \langle l_4, l_5, p' = p + m2\rangle$$

$$\begin{aligned}\tau_5 = \langle l_3, l_5, &(2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \neq 1) \vee (2 * \beta_3 = m1 \wedge m1 \leq 0 \wedge 0 \neq 1) \vee \\ &(2 * \beta_4 + 1 = m1 \wedge m1 \geq 0 \wedge 1 \neq 1) \vee (2 * \beta_4 - 1 = m1 \wedge m1 \leq 0 \wedge -1 \neq 1)\rangle\end{aligned}$$

$$\begin{aligned}\tau_6 = \langle l_5, l_6, &(2 * \beta_5 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_5) \vee (2 * \beta_5 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_5) \vee \\ &(2 * \beta_6 + 1 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_6) \vee (2 * \beta_6 - 1 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_6)\rangle\end{aligned}$$

$$\tau_7 = \langle l_6, l_2, m2' = 2 * m2\rangle$$

$$\tau_8 = \langle l_2, l_7, (m1 \leq 1)\rangle$$

Example 6. After applying step 3 and 4 we obtain the following Presburger transition system:

$$L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7\}, V = \{m1, m2, p, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$$

$$\tau_0 = \langle l_0, l_1, p' = 0\rangle$$

$$\tau_1 = \langle l_1, l_2, (m1 \geq 1) \wedge (m2 \geq 1)\rangle$$

$$\tau_2 = \langle l_2, l_3, (m1 > 0)\rangle$$

$$\begin{aligned}\tau_3 = \langle l_3, l_4, &(2 * \beta_1 = m1 \wedge m1 \geq 0 \wedge 0 = 1) \vee (2 * \beta_1 = m1 \wedge m1 \leq 0 \wedge 0 = 1) \vee \\ &(2 * \beta_2 + 1 = m1 \wedge m1 \geq 0 \wedge 1 = 1) \vee (2 * \beta_2 - 1 = m1 \wedge m1 \leq 0 \wedge -1 = 1)\rangle\end{aligned}$$

$$\tau_4 = \langle l_4, l_5, p' = p + m2\rangle$$

$$\begin{aligned}\tau_5 = \langle l_3, l_5, &(2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \leq 0) \vee (2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \geq 2) \vee \\ &(2 * \beta_3 = m1 \wedge m1 \leq 0 \wedge 0 \leq 0) \vee (2 * \beta_3 = m1 \wedge m1 \leq 0 \wedge 0 \geq 2) \vee \\ &(2 * \beta_4 + 1 = m1 \wedge m1 \geq 0 \wedge 1 \leq 0) \vee (2 * \beta_4 + 1 = m1 \wedge m1 \geq 0 \wedge 1 \leq 0) \vee \\ &(2 * \beta_4 - 1 = m1 \wedge m1 \leq 0 \wedge -1 \leq 0) \vee (2 * \beta_4 - 1 = m1 \wedge m1 \leq 0 \wedge 3 \leq 0)\rangle\end{aligned}$$

$$\begin{aligned}\tau_6 = \langle l_5, l_6, &(2 * \beta_5 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_5) \vee (2 * \beta_5 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_5) \vee \\ &(2 * \beta_6 + 1 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_6) \vee (2 * \beta_6 - 1 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_6)\rangle\end{aligned}$$

$$\tau_7 = \langle l_6, l_2, m2' = 2 * m2\rangle$$

$$\tau_8 = \langle l_2, l_7, (m1 \leq 1)\rangle$$

Example 7. Finally, once the transformation is completed after applying step 5, we have the next Presburger transition system:

$L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7\}, V = \{m1, m2, p, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$

$\tau_0 = \langle l_0, l_1, p' = 0\rangle$

$\tau_1 = \langle l_1, l_2, (m1 \geq 1) \wedge (m2 \geq 1)\rangle$

$\tau_2 = \langle l_2, l_3, (m1 > 0)\rangle$

$\tau_3 = \langle l_3, l_4, (2 * \beta_1 = m1 \wedge m1 \geq 0 \wedge 0 = 1)\rangle$

$\tau_4 = \langle l_3, l_4, (2 * \beta_1 = m1 \wedge m1 \leq 0 \wedge 0 = 1)\rangle$

$\tau_5 = \langle l_3, l_4, (2 * \beta_2 + 1 = m1 \wedge m1 \geq 0 \wedge 1 = 1)\rangle$

$\tau_6 = \langle l_3, l_4, (2 * \beta_2 - 1 = m1 \wedge m1 \leq 0 \wedge -1 = 1)\rangle$

$\tau_7 = \langle l_4, l_5, p' = p + m2\rangle$

$\tau_8 = \langle l_3, l_5, (2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \leq 0)\rangle$

$\tau_9 = \langle l_3, l_5, (2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \geq 2)\rangle$

$\tau_{10} = \langle l_3, l_5, (2 * \beta_3 = m1 \wedge m1 \leq 0 \wedge 0 \leq 0)\rangle$

$\tau_{11} = \langle l_3, l_5, (2 * \beta_3 = m1 \wedge m1 \leq 0 \wedge 0 \geq 2)\rangle$

$\tau_{12} = \langle l_3, l_5, (2 * \beta_4 + 1 = m1 \wedge m1 \geq 0 \wedge 1 \leq 0)\rangle$

$\tau_{13} = \langle l_3, l_5, (2 * \beta_4 + 1 = m1 \wedge m1 \geq 0 \wedge 1 \leq 0)\rangle$

$\tau_{14} = \langle l_3, l_5, (2 * \beta_4 - 1 = m1 \wedge m1 \leq 0 \wedge -1 \leq 0)\rangle$

$\tau_{15} = \langle l_3, l_5, (2 * \beta_4 - 1 = m1 \wedge m1 \leq 0 \wedge 3 \leq 0)\rangle$

$\tau_{16} = \langle l_5, l_6, (2 * \beta_5 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_5)\rangle$

$\tau_{17} = \langle l_5, l_6, (2 * \beta_5 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_5)\rangle$

$\tau_{18} = \langle l_5, l_6, (2 * \beta_6 + 1 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_6)\rangle$

$\tau_{19} = \langle l_5, l_6, (2 * \beta_6 - 1 = m1 \wedge m1 \leq 0 \wedge m1' = \beta_6)\rangle$

$\tau_{20} = \langle l_6, l_2, m2' = 2 * m2\rangle$

$\tau_{21} = \langle l_2, l_7, (m1 \leq 1)\rangle$

4 Generation of loop invariants

In this chapter, we describe an iterative method to generate loop linear invariants based on the technique presented in [1]. The strength of our approach lies in the use of problem encoding to avoid discovering linear combinations of already found invariants in each iteration. Besides, we detail other techniques to consider when generating invariants that improve the efficient or make possible to find some more types of invariants.

4.1. Inductive assertions and Farkas' Lemma

The main idea behind the technique explained in [1] is to represent a linear invariant

$$c_1x_1 + \cdots + c_nx_n + d \leq 0$$

in terms of unknown coefficients $c_1, \ldots, c_n, d$ and generate constraints on the coefficients such that any solution corresponds to an inductive assertion.

An assertion is said to be *inductive* at a program location if it holds the first time the location is reached and is preserved under every cycle back to the location. It has been established that all inductive assertions are invariant. Furthermore, the standard method for proving a given assertion invariant is to find an inductive assertion that strengthens it.

The key to this approach is Farkas' lemma which provides a sound and complete method for reasoning about systems of linear inequalities.

Theorem 1 (Farkas' Lemma). *Consider the following system of linear inequalities over real-valued variables $x_1, \cdots, x_n$*

$$S : \begin{bmatrix} a_{11}\,x_1 + \cdots + a_{1n}\,x_n + b_1 \leq 0 \\ \vdots \qquad\qquad \vdots \quad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n + b_m \leq 0 \end{bmatrix}$$

When S is satisfiable, it entails a given linear inequality

$$\psi : c_1x_1 + \cdots + c_nx_n + d \leq 0$$

if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_m$, such that

$$c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \quad \ldots \quad , c_n = \sum_{i=1}^{m} \lambda_i a_{in},\ d = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

Furthermore, S is unsatisfiable if and only if the inequality $1 \leq 0$ can be derived as shown above.

We represent applications of the lemma using a tabular notation:

$$\begin{array}{c|rcl}
\lambda_0 & & -1 \le 0 & \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + & b_1 \le 0 & \left.\vphantom{\begin{array}{c}1\\1\\1\end{array}}\right\} S \\
\vdots & \vdots \qquad\qquad \vdots & \vdots & \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + & b_m \le 0 & \\
\hline
 & c_1x_1 + \cdots + c_nx_n + & d \le 0 & \leftarrow \psi \\
 & & 1 \le 0 & \leftarrow false
\end{array}$$

The antecedents are placed above the line and the consequences below. For each column, the sum of the column entries above the line, with the appropriate multipliers, must be equal to the entry below the line. If a row corresponds to an inequality, the corresponding multiplier is required to be non-negative. This requirement is dropped for rows corresponding to equalities.

4.2. Inductive assertion maps and Problem encoding

Definition 5 (Inductive Assertion Map) Given a program P with a cutset C and an assertion $\eta_c(l)$, for each cutpoint l, we say that η_c is an *inductive assertion map* for C if it satisfies the following conditions for all cutpoints l, l':

Initiation For each basic path π from l_0 to l, $\rho_\pi \models \eta_c(l)'$.

Consecution For each basic path π from l to l', $\eta_c(l) \wedge \rho_\pi \models \eta_c(l')'$.

Given a transition system and a cutset, we generate a (partial) inductive assertion map η over the cutpoints by encoding initiation and consecution. Let $\eta(l)$ be represented by the assertion $c_{l_1}x_1 + \cdots + c_{l_n}x_n + d_l \le 0$, where each c_{l_i} and each d_l is an unknown. The two conditions for the map to be inductive are encoded as follows:

Initiation. For each cutpoint l_j and each basic path π from l_0 to l_j, the path may be an enabled path, in which case ρ_π is satisfiable, or the path may be disabled, in which case, ρ_π is unsatisfiable. Initiation can thus be represented by the following table,

$$\begin{array}{c|rcl}
\lambda_0 & & -1 \le 0 & \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n + & b_1 \le 0 & \left.\vphantom{\begin{array}{c}1\\1\\1\end{array}}\right\} \rho_\pi \\
\vdots & \vdots \qquad\qquad \vdots \qquad \vdots \qquad\qquad \vdots & \vdots & \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + & b_m \le 0 & \\
\hline
 & c_{l_j1}x'_1 + \cdots + c_{l_jn}x'_n + & d_{l_j} \le 0 & \leftarrow \eta(l_j)' \\
 & & 1 \le 0 & \leftarrow disabled
\end{array}$$

where $\lambda_0, \ldots, \lambda_m \geq 0$.

Consecution. For each basic path π from a cutpoint l_i to a cutpoint l_j, we encode the consecution condition, $\eta(l_i) \wedge \rho_\pi \models \eta(l_j)'$, using Farkas' Lemma. The path may be an enabled path, in which case $\eta(l_i) \wedge \rho_\pi$ is satisfiable, or the path may be disabled, in which case, $\eta(l_i) \wedge \rho_\pi$ is unsatisfiable.

The constraints are represented by the table shown below:

$$\begin{array}{c|rcl}
\mu & c_{l_i1}x_1 + \cdots + c_{l_in}x_n & + d_{l_i} \leq 0 & \leftarrow \eta(l_i) \\
\lambda_0 & & -1 \leq 0 & \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n & + b_1 \leq 0 & \multirow{3}{*}{$\left.\vphantom{\begin{array}{c}a\\ \vdots\\ a\end{array}}\right\} \rho_\pi$} \\
\vdots & \vdots \qquad \vdots \qquad \vdots \qquad \vdots & \vdots & \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n & + b_m \leq 0 & \\
\hline
 & c_{l_j1}x'_1 + \cdots + c_{l_jn}x'_n & + d_{l_j} \leq 0 & \leftarrow \eta(l_j)' \\
 & & 1 \leq 0 & \leftarrow \textit{disabled}
\end{array}$$

where $\mu, \lambda_0, \ldots, \lambda_m \geq 0$.

The constraints corresponding to initiation are linear, as are the constraints corresponding to the disabled case of consecution. However, the constraints for the enabled case of consecution are non-linear due to the presence of the multiplier μ in a row containing unknown coefficients.

4.3. Set of invariants for a cutpoint

Given a transition system and a cutset $C = \{p_1, \ldots, p_r\}$, let $\Pi = \{\pi_1, \ldots, \pi_m\}$ be the set of basic paths from $l_i \in \{l_0\} \cup C$ to $l_j \in C$ and let $\mathcal{E}(\pi_k)$ and $\mathcal{D}(\pi_k)$ be the constraints for π_k corresponding to the enabled and disabled cases respectively described in the previous section. Then, the problem of finding an inductive assertion map is encoded on the constraint:

$$(\exists \bar{c}_{p_1}, d_{p_1}, \ldots, \bar{c}_{p_r}, d_{p_r})(\mathcal{C}(\pi_1) \wedge \ldots \wedge \mathcal{C}(\pi_m))$$

where $\bar{c}_{p_j}, d_{p_j}$ are the coefficients for cutpoint p_j and $\mathcal{C}(\pi_k) = \mathcal{E}(\pi_k) \vee \mathcal{D}(\pi_k)$. The solution to that problem, if any, give us one invariant for each cutpoint. Suppose we have obtained the invariant $\varphi_{p0} : C_{p1}x_1 + \cdots + C_{pn}x_n + D_p \leq 0$ for the cutpoint p. We can find a different invariant for p setting out the problem:

$$(\exists \bar{c}_{p_1}, d_{p_1}, \ldots, \bar{c}_{p_r}, d_{p_r})(\neg(c_{p0} = C_{p0} \wedge \ldots \wedge c_{pn} = C_{pn}) \wedge \mathcal{C}(\pi_1) \wedge \ldots \wedge \mathcal{C}(\pi_m))$$

Nevertheless, the new obtained invariant may be a consequence of the first one, and hence useless. A better approach is to avoid the new invariant to be a logical consequence of the previous one, i.e., asserting $\neg(\varphi_{p0} \rightarrow \varphi_p) \equiv \neg(\neg\varphi_{p0} \vee \varphi_p) \equiv (\varphi_{p0} \wedge \neg\varphi_p)$ where φ_p is the target invariant $c_{p1}x_1 + \cdots + c_{pn}x_n + d_p \leq 0$ (see lemma 1). Besides, we can avoid invariants which do not contain any variable forcing $(\overline{c}_p \neq \overline{0})$, that is, $(c_{p1} \neq 0 \vee \ldots \vee c_{pn} \neq 0)$.

In general, when we had discovered k invariants $\varphi_{p0}, \ldots, \varphi_{pk}$ for a cutpoint p we have to set out the problem:

$$(\exists \overline{c}_{p_1}, d_{p_1}, \ldots, \overline{c}_{p_r}, d_{p_r})((\overline{c}_p \neq \overline{0}) \wedge (\varphi_{p0} \wedge \ldots \wedge \varphi_{pk} \wedge \neg\varphi_p) \wedge \mathcal{C}(\pi_1) \wedge \ldots \wedge \mathcal{C}(\pi_m))$$

Remark that φ_p is a non-linear expression, which is the second source of non-linearity in the formula generated (recall the other one is owing to multiplier μ).

Notice also that after finishing the discovery process for a cutpoint, some of the last found invariants can entail a subset of the first ones because it is stronger. In order to minimize the set of invariants $\{\varphi_{p0}, \ldots, \varphi_{pq}\}$ for a cutpoint p, we must check for every $k \in \{0, \ldots, q\}$ whether an invariant φ_{pk} is a logical consequence of the rest and it can be removed, i.e, if the next (only linear arithmetic) constraint problem is unsatisfiable:

$$(\exists \overline{x})(\varphi_{p0} \wedge \ldots \wedge \varphi_{pr} \wedge \neg\varphi_{pk})$$

4.4. Cutset choice

There are many ways of choosing a valid cutset. In CppInv the valid cutset is built by establishing as cutpoints every location associated to a label which is destination of a `goto` jump that appears at a later location than the label point and every entry location of a `while` or `do-while` loop. This method is simple and effective but it might select an excessive number of cutpoints. With the aim of reducing the cardinality of the cutset, it is possible to remove every cutpoint which has no basic back path to itself. However, even with this simplification the obtained cutset can not be ensured to be any optimal cutset, i.e., one which contains a minimal number of cutpoints necessary for covering every cycle. On the other hand, note that sometimes instead of a optimal cutset we might be interested in an optimal cutset containing some selected subset of program locations. The study and implementation of theses alternatives and other ones is left out of this thesis scope.

Example 8. By way of illustration, a cutset, which has been selected following the procedure that is executed by the tool, can be seen in Example 12. Notice that, in that case, since every back path to cutpoint l_3 passes through cutpoints l_4 and l_6, simplification can be applied removing l_3.

4.5. Elimination of unfeasible paths

If we know that the relation associated with a path is unsatisfiable, we can remove it. This simplification has two advantages. On the one hand, we decrease the number of constraints of the SMT problem which is, in general, better for the SMT-solver. And on the other hand, if every path which reaches a cutpoint is discarded, then we can mark the cutpoint as unreachable and remove all paths with that cutpoint as starting point. Notice that invariants for an unreachable cutpoint have no sense and, therefore, we can avoid generating them.

Example 9. Consider the final Presburger transition system of Example 7. All cycles of the program are cut by l_2 which is the entry of a `while` loop in the original source code. The only feasible paths are the initiation path π_1 from l_0 to l_2 using (τ_0, τ_1), and the consecution paths π_2 and π_3 cycle back to l_2 through $(\tau_2, \tau_5, \tau_7, \tau_{18}, \tau_{20})$ and $(\tau_2, \tau_8, \tau_{16}, \tau_{20})$ repectively. The relations associated with these paths are:

$$\boxed{\begin{aligned}
&\rho_{\pi_1} : p' = 0 \wedge (m1 \geq 1) \wedge (m2 \geq 1) \wedge m1' = m1 \wedge m2' = m2\\
&\rho_{\pi_2} : (m1 > 0) \wedge (2 * \beta_2 + 1 = m1 \wedge m1 \geq 0 \wedge 1 = 1) \wedge p' = p + m2 \wedge\\
&\qquad (2 * \beta_6 + 1 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_6) \wedge m2' = 2 * m2\\
&\rho_{\pi_3} : (m1 > 0) \wedge (2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \leq 0) \wedge p' = p + m2 \wedge\\
&\qquad (2 * \beta_5 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_5) \wedge m2' = 2 * m2
\end{aligned}}$$

4.6. Extensions of linear invariants

The presented generation method can discover linear inductive assertions over program variables. However, sometimes it is necessary to find an invariant which state a relationship between program variables and its unknown initial values. We can obtain this kind of invariants adding auxiliary variables whose values are not changed throughout the program and are assigned to the corresponding variables when those are declared.

Another interesting extension is the generation of non-linear inductive assertions. As done for instance in [5], a simple way of obtaining them consists of adding a new auxiliary variable for each monomial that is not a single variable and conditions about its value. Being precise, for every path encoded as a constraint, the value of the new variable must be asserted according to the values of the variables that forms the monomial in that path.

Notice that in order to define the value of a monomial, it might be necessary to introduce new auxiliary variables that represent other monomials which are not considered previously. We can ensure the procedure finishes after a finite number of steps because accepted programs can only contain linear expressions and, therefore, a finite number of monomials with degree equal to or less than the current monomial degree can be generated.

CppInv is able to add auxiliary variables that represent the initial value of uninitialized variables and introduce quadratic monomials for a given list of pairs of variable names in a automic way.

Example 10. Next figure shows how Example 3 would have looked if we had written the code including auxiliary variables m and M to represent respectively the monomial $m1m2$ and its initial value:

```
int main() {
  // Pre: m1=M1 /\ m2=M2
  int m1, m2, p=0;
  // m: m1*m2, M: M1*M2
  int m=M;
  assert(m1>=1 && m2>=1);
  while (m1>0) {
    if (m1%2==1) {
      p=p+m2;
      m=m-m2;
    }
    m1=m1/2;
    m2=2*m2;
  }
  // Post: p = M1*M2;
}
```

Notice how variable m is updated in function of $m1'$ and $m2'$ realizing that:

$$m' = m1'{*}m2' = \begin{cases} \dfrac{m1}{2} * 2m2 = m1 * m2 = m, & \text{if } (m1 \bmod 2 = 0) \\ \dfrac{m1-1}{2} * 2m2 = m1 * m2 - m2 = m - m2, & \text{if } (m1 \bmod 2 = 1) \end{cases}$$

The modified version of the running example has also $C = \{l_2\}$ as cutset. Moreover, the paths π_1, π_2 and π_3 shown in Example 9 are applicable except for its relations, which now assert also about M' and m'. Specifically, we show bellow changes bolded:

$$\boxed{\begin{array}{l}
\rho_{\pi_1} : p' = 0 \wedge (m1 \geq 1) \wedge (m2 \geq 1) \wedge m1' = m1 \wedge m2' = m2 \wedge \mathbf{M}' = \mathbf{M} \wedge \mathbf{m}' = \mathbf{M} \\
\rho_{\pi_2} : (m1 > 0) \wedge (2 * \beta_2 + 1 = m1 \wedge m1 \geq 0 \wedge 1 = 1) \wedge p' = p + m2 \wedge \\
\qquad (2 * \beta_6 + 1 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_6) \wedge m2' = 2 * m2 \wedge \mathbf{M}' = \mathbf{M} \wedge \mathbf{m}' = \mathbf{m} - \mathbf{m2} \\
\rho_{\pi_3} : (m1 > 0) \wedge (2 * \beta_3 = m1 \wedge m1 \geq 0 \wedge 0 \leq 0) \wedge p' = p + m2 \wedge \\
\qquad (2 * \beta_5 = m1 \wedge m1 \geq 0 \wedge m1' = \beta_5) \wedge m2' = 2 * m2 \wedge \mathbf{M}' = \mathbf{M} \wedge \mathbf{m}' = \mathbf{m}
\end{array}}$$

Let $\varphi : c_1 m1 + c_2 m2 + c_3 p + c_4 M + c_5 m + d \leq 0$ be the target invariant at l_2. Then the path π_1 generates the constraints given by the following table:

$$\begin{array}{r|lllllllr}
\lambda_0 & & & & & & & & -1 \leq 0 \\
\lambda_1 & -m1 & & & & & & & +1 \leq 0 \\
\lambda_2 & & -m2 & & & & & & +1 \leq 0 \\
\lambda_3 & m1 & & -\ m1' & & & & & = 0 \\
\lambda_4 & & m2 & & -\ m2' & & & & = 0 \\
\lambda_5 & & & & & p' & & & = 0 \\
\lambda_6 & & M & & & & -\ M' & & = 0 \\
\lambda_7 & & M & & & & & -\ m' & = 0 \\
\hline
 & \multicolumn{8}{r}{c_1 m1' + c_2 m2' + c_3 p' + c_4 M' + c_5 m' + d \leq 0} \\
 & \multicolumn{8}{r}{1 \leq 0}
\end{array}$$

resulting in the next constraints for the enabled case:

$$\mathcal{E}(\pi_1) = (\exists \overline{\lambda}) \left[\begin{pmatrix} \lambda_0 \geq 0 \wedge \\ \lambda_1 \geq 0 \wedge \\ \lambda_2 \geq 0 \wedge \end{pmatrix} \wedge \begin{pmatrix} 0 = -\lambda_1 + \lambda_3 \wedge 0 = -\lambda_2 + \lambda_4 \wedge 0 = \lambda_6 + \lambda_7 \wedge \\ c_1 = -\lambda_3 \wedge c_2 = -\lambda_4 \wedge c_3 = -\lambda_5 \wedge c_4 = -\lambda_6 \wedge \\ c_5 = -\lambda_7 \wedge d = -\lambda_0 + \lambda_1 + \lambda_2 \end{pmatrix} \right]$$

and the next constraints for the disabled case:

$$\mathcal{D}(\pi_1) = (\exists \overline{\lambda}) \left[\begin{pmatrix} \lambda_0 \geq 0 \wedge \\ \lambda_1 \geq 0 \wedge \\ \lambda_2 \geq 0 \wedge \end{pmatrix} \wedge \begin{pmatrix} 0 = -\lambda_1 + \lambda_3 \wedge 0 = -\lambda_2 + \lambda_4 \wedge 0 = \lambda_6 + \lambda_7 \wedge \\ 0 = -\lambda_3 \wedge 0 = -\lambda_4 \wedge 0 = -\lambda_5 \wedge 0 = -\lambda_6 \wedge \\ 0 = -\lambda_7 \wedge 0 = -\lambda_0 + \lambda_1 + \lambda_2 \end{pmatrix} \right]$$

The path π_2 generates the constraints given by the following table:

$$
\begin{array}{c|l}
\mu & c_1m1 + c_2m2 + c_3p + c_4M + c_5m \quad + d \leq 0 \\
\lambda_0 & -1 \leq 0 \\
\lambda_1 & -\ m1 \quad +1 \leq 0 \\
\lambda_2 & -\ m1 \quad + 2\beta_2 \quad +1 = 0 \\
\lambda_3 & -\ m1 \quad \leq 0 \\
\lambda_5 & -\ m1 \quad + 2\beta_6 \quad +1 = 0 \\
\lambda_6 & -\ m1 \quad \leq 0 \\
\lambda_7 & \beta_6 - \ m1' \quad = 0 \\
\lambda_8 & 2m2 \quad -\ m2' \quad = 0 \\
\lambda_9 & m2 + \ p \quad -\ p' \quad = 0 \\
\lambda_{10} & M \quad -\ M' \quad = 0 \\
\lambda_{11} & -\ m2 \quad + \ m \quad -\ m' \quad = 0 \\
\hline
 & c_1m1' + c_2m2' + c_3p' + c_4M' + c_5m' + d \leq 0 \\
 & 1 \leq 0
\end{array}
$$

resulting in the next constraints for the enabled case:

$$
\mathcal{E}(\pi_2) = (\exists \mu, \overline{\lambda}) \begin{pmatrix} \mu \ \geq 0 \ \wedge \\ \lambda_0 \geq 0 \ \wedge \\ \lambda_1 \geq 0 \ \wedge \\ \lambda_3 \geq 0 \ \wedge \\ \lambda_6 \geq 0 \end{pmatrix} \wedge \begin{pmatrix} 0 = \mu * c_1 - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_5 - \lambda_6 \ \wedge \\ 0 = \mu * c_2 + 2\lambda_8 + \lambda_9 - \lambda_{11} \ \wedge \\ 0 = \mu * c_3 + \lambda_9 \wedge 0 = \mu * c_4 + \lambda_{10} \ \wedge \\ 0 = \mu * c_5 + \lambda_{11} \wedge 0 = 2\lambda_2 \ \wedge \\ 0 = 2\lambda_5 + \lambda_7 \wedge c_1 = -\lambda_7 \wedge c_2 = -\lambda_8 \ \wedge \\ c_3 = -\lambda_9 \wedge c_4 = -\lambda_{10} \wedge c_5 = -\lambda_{11} \ \wedge \\ d = \mu * d - \lambda_0 + \lambda_1 + \lambda_2 + \lambda_5 \end{pmatrix}
$$

and the next constraints for the disabled case:

$$
\mathcal{D}(\pi_2) = (\exists \mu, \overline{\lambda}) \begin{pmatrix} \mu \ \geq 0 \ \wedge \\ \lambda_0 \geq 0 \ \wedge \\ \lambda_1 \geq 0 \ \wedge \\ \lambda_3 \geq 0 \ \wedge \\ \lambda_6 \geq 0 \end{pmatrix} \wedge \begin{pmatrix} 0 = \mu * c_1 - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_5 - \lambda_6 \ \wedge \\ 0 = \mu * c_2 + 2\lambda_8 + \lambda_9 - \lambda_{11} \ \wedge \\ 0 = \mu * c_3 + \lambda_9 \wedge 0 = \mu * c_4 + \lambda_{10} \ \wedge \\ 0 = \mu * c_5 + \lambda_{11} \wedge 0 = 2\lambda_2 \ \wedge \\ 0 = 2\lambda_5 + \lambda_7 \wedge 0 = -\lambda_7 \wedge 0 = -\lambda_8 \ \wedge \\ 0 = -\lambda_9 \wedge 0 = -\lambda_{10} \wedge c0 = -\lambda_{11} \ \wedge \\ 1 = \mu * d - \lambda_0 + \lambda_1 + \lambda_2 + \lambda_5 \end{pmatrix}
$$

And finally, the path π_3 generates the constraints given by the following table:

μ	c_1m1+	c_2m2+	c_3p+	c_4M+	c_5m								$+d\leq 0$
λ_0													$-1\leq 0$
λ_1	$-\ m1$												$+1\leq 0$
λ_2	$-\ m1$					$+2\beta_3$							$=0$
λ_3	$-\ m1$												≤ 0
λ_5	$-\ m1$						$+2\beta_5$						$=0$
λ_6	$-\ m1$												≤ 0
λ_7							β_5-	$m1'$					$=0$
λ_8		$2m2$							$-\ m2'$				$=0$
λ_9			p							$-\ p'$			$=0$
λ_{10}				M							$-\ M'$		$=0$
λ_{11}					m							$-\ m'$	$=0$
								$c_1m1'+$	$c_2m2'+$	$c_3p'+$	$c_4M'+$	c_5m'	$+d\leq 0$
													$1\leq 0$

resulting in the next constraints for the enabled case:

$$\mathcal{E}(\pi_3) = (\exists\mu,\overline{\lambda}) \begin{pmatrix} \mu\ \geq 0\ \wedge \\ \lambda_0 \geq 0\ \wedge \\ \lambda_1 \geq 0\ \wedge \\ \lambda_3 \geq 0\ \wedge \\ \lambda_6 \geq 0 \end{pmatrix} \wedge \begin{pmatrix} 0 = \mu * c_1 - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_5 - \lambda_6\ \wedge \\ 0 = \mu * c_2 + 2\lambda_8\ \wedge \\ 0 = \mu * c_3 + \lambda_9 \wedge 0 = \mu * c_4 + \lambda_{10}\ \wedge \\ 0 = \mu * c_5 + \lambda_{11} \wedge 0 = 2\lambda_2\ \wedge \\ 0 = 2\lambda_5 + \lambda_7 \wedge c_1 = -\lambda_7 \wedge c_2 = -\lambda_8\ \wedge \\ c_3 = -\lambda_9 \wedge c_4 = -\lambda_{10} \wedge c_5 = -\lambda_{11}\ \wedge \\ d = \mu * d - \lambda_0 + \lambda_1 \end{pmatrix}$$

and the next constraints for the disabled case:

$$\mathcal{D}(\pi_3) = (\exists\mu,\overline{\lambda}) \begin{pmatrix} \mu\ \geq 0\ \wedge \\ \lambda_0 \geq 0\ \wedge \\ \lambda_1 \geq 0\ \wedge \\ \lambda_3 \geq 0\ \wedge \\ \lambda_6 \geq 0 \end{pmatrix} \wedge \begin{pmatrix} 0 = \mu * c_1 - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_5 - \lambda_6\ \wedge \\ 0 = \mu * c_2 + 2\lambda_8\ \wedge \\ 0 = \mu * c_3 + \lambda_9 \wedge 0 = \mu * c_4 + \lambda_{10}\ \wedge \\ 0 = \mu * c_5 + \lambda_{11} \wedge 0 = 2\lambda_2\ \wedge \\ 0 = 2\lambda_5 + \lambda_7 \wedge 0 = -\lambda_7 \wedge 0 = -\lambda_8\ \wedge \\ 0 = -\lambda_9 \wedge 0 = -\lambda_{10} \wedge 0 = -\lambda_{11}\ \wedge \\ 1 = \mu * d - \lambda_0 + \lambda_1 \end{pmatrix}$$

Given the previous constraints we can set out the problem:

$$(\exists\overline{c}, d)((\overline{c} \neq \overline{0}) \wedge \mathcal{C}(\pi_1) \wedge \mathcal{C}(\pi_2) \wedge \mathcal{C}(\pi_3))$$

After solving, we obtain the solution $c_1 = c_2 = d = 0$, $c_3 = c_5 = -1$ and $c_4 = 1$ that corresponds to the invariant $\varphi_0 : -p + M - m \leq 0$.

Applying the technique explained in section 4.3, we obtain the following additional invariants $\varphi_1 : -m2 + 1 \leq 0$, $\varphi_2 : p - M + m \leq 0$ and $\varphi_3 : -2m1 \leq 0$. No invariant is discarded because no invariant is logical consequence of the others.

Notice that $\varphi_0 \wedge \varphi_2 \Rightarrow p = M - m$, which can be used to prove partial correctness of the algorithm.

4.7. Adding invariants to path relations

The method for generating invariants is inductive. That means, we can use the discovered invariance information as feedback to the assertions associated with paths.

Once a set of invariants is generated for a cutpoint, we supply them to every basic path which has the cutpoint as source. This might cause some of the paths become disabled and, in that case, they are discarded. Futhermore, if there are no more paths that reach the destination of some eliminated path, the destination must be marked as unreachable and all paths which has that location as source removed.

After finishing the elimination process, new solutions can be generated from the resulting paths repeating the procedure explained in section 4.3.

Example 11. We can now continue with Example 10 checking what happens after adding discovered invariants φ_0, φ_1, φ_2 and φ_3 to the relations of the consecution paths π_2 and π_3. Let π_2' and π_3' be the paths π_2 and π_3, repectively, with the new invariance information, i.e., for $i \in \{2, 3\}$:

$$\rho_{\pi_i'} = \rho_{\pi_i} \wedge \varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3$$

Since $\rho_{\pi_2'}$ and $\rho_{\pi_3'}$ remain satisfiable, new constraints are obtained for those paths. By way of illustration, we shows how the common part of the tables representing the constraints for π_2' and π_3' looks:

$$\begin{array}{r|lllll r r}
\mu & c_1 m1 + & c_2 m2 + & c_3 p + & c_4 M + & c_5 m & & +d \le 0 \\
\lambda_0 & & & & & & & -1 \le 0 \\
\vdots & & & \vdots & & & \vdots & \vdots\ 0 \\
\lambda_{12} & & & -\ p + & M - & m & & \le 0 \\
\lambda_{13} & & -\ m2 & & & & & +1 \le 0 \\
\lambda_{14} & & & p - & M + & m & & \le 0 \\
\lambda_{15} & -\ 2m1 & & & & & & \le 0 \\
\hline
 & & & & & & c_1 m1' + c_2 m2' + c_3 p' + c_4 M' + c_5 m' & +d \le 0 \\
 & & & & & & & 1 \le 0
\end{array}$$

In addition to the new generated constraints $\mathcal{E}(\pi_2')$, $\mathcal{D}(\pi_2')$, $\mathcal{E}(\pi_3')$ and $\mathcal{D}(\pi_3')$, the new SMT problem must also encode that the target invariant can not be logical consequence of the already found invariants. In particular, we start setting out the problem:

$$(\exists \bar{c}, d)((\bar{c} \neq \bar{0}) \wedge \mathcal{C}(\pi_1) \wedge \mathcal{C}(\pi_2') \wedge \mathcal{C}(\pi_3') \wedge \varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \neg\varphi)$$

whose solution is $\varphi_0' : -m2 + p \leq 0$. Following again the procedure explained in section 4.3, we also obtain $\varphi_1' : -m2 + p + 1 \leq 0$, $\varphi_2' : -5p + M - m \leq 0$ and $\varphi_3' : -2m1 - m2 + 5p - 5M + 5m + 2 \leq 0$.

In order to check whether φ_0 is a logical consenquence of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_0' \wedge \varphi_1' \wedge \varphi_2' \wedge \varphi_3'$, we set out the problem:

$$\neg\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_0' \wedge \varphi_1' \wedge \varphi_2' \wedge \varphi_3'$$

Since the problem is satisfiable, φ_0 is not entailed by the other invariants. The same happens with the rest of invariants except for φ_1 and φ_0' which are discarded because they are detected as logical consequences.

Repeating the process after adding the new set of invariants, we obtain $\varphi_0'' : -4m1 - m2 - 3p + 3 \leq 0$, $\varphi_1'' : -6m1 - m2 - 6p - M + m + 4 \leq 0$ and $\varphi_2'' : -2m1 - 6p - 6M + 6m + 1 \leq 0$. In this case, φ_3', φ_0'' and φ_2'' are identified as logical consequences of the total set of invariants and are dropped.

Next iteration does not find new inductive assertions. Therefore, the final set of invariants for l_2 is $\{\varphi_0, \varphi_2, \varphi_3, \varphi_1', \varphi_2', \varphi_1''\}$.

4.8. Cutpoints dependencies

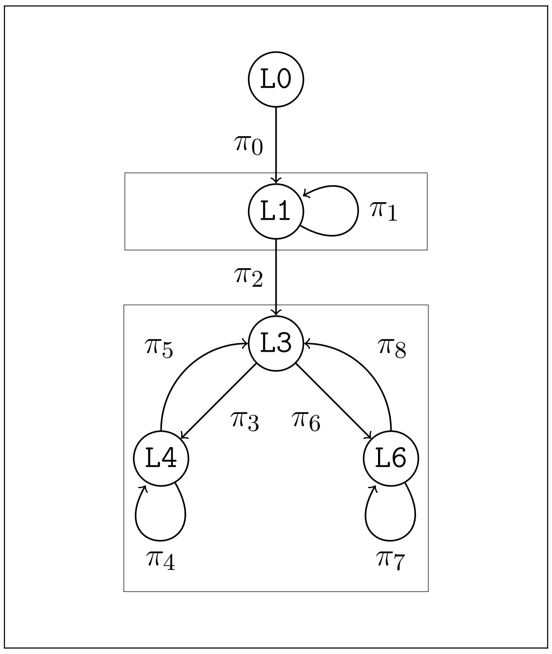
Invariant generation for a cutpoint usually depends on the existence of other cutpoints' invariants. These dependencies might be sequential or cyclical. We say a sequential dependency ocurrs when a cutpoint depends on some invariant of other cutpoint but invariants of the latter are not affected, directly or undirectly through other cutpoints, by any invariant of the former. On the other hand, a cyclical dependency ocurrs when somehow both cutpoints depend on each other.

Example 12. Consider the next labeled program together with its control flow graph induced by the initial location l_0 and the cutset $C = \{l_1, l_3, l_4, l_6\}$:

Notice that a simplification of the cutset C is possible (see Example 8). However, for the sake of clarity, the whole initial cutset is used for the next explanations. Now consider an inductive assertion map $\eta_c(l)$ for C. As it was said in section 4.2, the following conditions have to be satisfied:

$$\begin{array}{rrr} \rho_{\pi_0} \models \eta_c(l_1)', & \eta_c(l_1) \wedge \rho_{\pi_1} \models \eta_c(l_1)', & \eta_c(l_1) \wedge \rho_{\pi_2} \models \eta_c(l_3)' \\ \eta_c(l_3) \wedge \rho_{\pi_3} \models \eta_c(l_4)', & \eta_c(l_4) \wedge \rho_{\pi_4} \models \eta_c(l_4)', & \eta_c(l_4) \wedge \rho_{\pi_5} \models \eta_c(l_3)' \\ \eta_c(l_3) \wedge \rho_{\pi_6} \models \eta_c(l_6)', & \eta_c(l_6) \wedge \rho_{\pi_7} \models \eta_c(l_6)', & \eta_c(l_6) \wedge \rho_{\pi_8} \models \eta_c(l_8)' \end{array}$$

```
int main()
{
L0: int i=1, j;
L1: while (i<=j) {
L2:   i=5*i+1;
    }
    do {
L3:   if (i<j) {
L4:     while (i<j) {
L5:       i=i*2+1;
        }
      } else {
L6:      while (j<=i) {
L7:        j=j*3+1;
         }
      }
   }  while (true);
}
```



Since $\eta_c(l_3)$ depends on $\eta_c(l_1)$ but not on the other way around, there exists a sequential dependency of l_3 invariants which are subjected to l_1 invariants. In addition, there is a cyclical dependency that affects l_3, l_4 and l_6 using the paths π_3, π_4, π_5, π_6, π_7 and π_8.

The advantage of sequential dependencies is that these ones give us an order to generate independently the set of invariants of each cutpoint, thus, reducing the number of constraints of the SMT problems to solve. Furthermore, since all invariants needed to generate a subsequent invariant have been calculated in advance, the non-linear constraints, owing to consecution encoding, are converted into linear constraints due to they are now concrete invariants instead of template invariants.

Example 13. Continuing with Example 12, notice that we can generate the set of invariants of l_1 independently of l_3 and once we have it, suppose it is $\{\varphi_{10}, \ldots, \varphi_{1m}\}$, the condition that must be satisfied now is $\varphi_{10} \wedge \ldots \wedge \varphi_{1m} \wedge \rho_{\pi_2} \models \eta_c(l_3)'$ which generates only linear constraints because the $\eta_c(l_1)$ template invariant does not appear any more.

In order to identify cutpoints' dependencies, we descompose the control flow graph of a program in its strongly connected components. Recall that if each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, which is called the condensation of the graph. Due to the condesantion is acyclic, there exists at least one topological ordering of its vertices.

Therefore, finding some sorting on the strongly connected components is equivalent to discovering a sequential dependency order on the set of subsets of cyclical dependent cutpoints.

Since we want to find a descomposition of the control flow graph and a topological ordering, we use the Tarjan algorithm which has the next useful property. The order in which the strongly connected components are identified constitutes a reverse topological sort of the condensation of the graph.

Example 14. The squared zones of the control flow graph shown in Example 12 corresponds to the strongly connected components of the graph (including some cutpoint), that is, $SCC_1 = \{l_1\}$ and $SCC_2 = \{l_3, l_4, l_6\}$. A topological sorting is the sequence (SCC_1, SCC_2).

Figure 4.1 shows in pseudo-code the main algorithm to find invariants taking into account dependencies. For each strongly connected component, it tries to discover new sets of invariants for pending cutpoints. If the invariants generated for a cutpoint affects other cutpoints in the same strongly connected component, they are mark as pending. This procedure is repeated until no more invariants are found for pending cutpoints.

Recall that new invariants may disable some paths. And if a disabled path was the last path connecting the destination cutpoint, this cutpoint becomes unreachable.

```
type Result = Set<Cutpoint,Pair<IsReachable,Invariants>>;

procedure findAllProgramInvariants(Graph G) return Result
{
  Result result;
  Stack<List<Cutpoint>> SCC = tarjan(G, result);
  while (!SCC.empty())
  {
    comp = SCC.pop(); // retrieved in topological order
    Stack<Cutpoint> pending_cps(comp); // initialize with SCC cutpoints
    while (!pending_cps.empty())
    {
      cp = pending_cps.pop();
      if (is_reachable(cp, result))
      {
        new_invs = findCutpointInvariants(cp);
        simplify(new_invs, result);
        addInvariants(new_invs, cp, result, G);
        markPendingAffectedCutpoints(pending_cps, cp, result, G);
      }
    }
  }
  return result;
}
```

Figure 4.1 The main algorithm to find invariants

5 Tool implementation

Next sections contain some comments related to the particular implementation of our tool.

5.1. Environment and dependencies

CppInv has been run and tested on a Linux platform. In order to solve the constraint problems obtained by the techniques explained in Chapter 4, the tool uses the SMT solver Barcelogic [3]. Recall that the reason for chosing Barcelogic was motivated by its ability to prove efficiently satisfiability of boolean formulas over non-linear integer arithmetic. Nevertheless, CppInv can be easily adapted to any other solver which handles the same logic and accepts the SMT-LIB standard language [15] as input.

5.2. Tool function and usage

The behaviour of CppInv is as follows. The tool reads a program, written in the input language specified in Section 3.1, from the standard input or a file (if some one is given as the last command line argument) and outputs the transition system that represents the program, the feasible execution paths for the initiation and consecution cases, and a set of invariants for each selected cutpoint of the program. Besides, CppInv can take several parameters which allows users to adjust SMT solving configuration options, and enhance or modify the program modeling information used for invariant generation. Next figure shows the help message of the tool where the accepted parameters are listed:

```
USAGE: ./cppinv [options] [FILENAME]
OPTIONS:
  --help              Display this information

  -save-files         Save intermediate SMT input
  -tlimit <sec>       Set <sec> seconds of timeout for solving
  -maxc <val>         Set the maximum value for which it finds coefficients

  -init-vars          Add variables that represent initial values
  -QM <var1> <var2>   Add a variable that represents quadratic
                      monomial var1*var2
  -no-min-cutset      Do not execute minimization of cutset
```

The `-save-files` option stores on the same directory where is the executable the SMT problems that are sent to Barcelogic. It has only informative purpose.

The `-init-vars` and `-QM` options corresponds to the extensions explained in Section **4.6**. The `-QM` option can be used as many times as necessary as it can be observed in Examples **A.1** and **A.3**. And the `-no-min-cutset` option avoid the simplification described in Section **4.4**.

Finally, the `-tlimit` and `-maxc` options are used as criteria paremeters for the termination of the SMT solver. Recall that Barcelogic can prove efficiently satisfiability of non-linear arithmetic constraints due to incompleteness can not handle many unsatisfiable problems. Therefore, it is necessary to enforce some limit. In the case of the solver used the possible limits are: setting a timeout or fixing a maximum range value for which it must find coefficients.

On the one hand, we have the second criterion that is more ‘deterministic’ and works very well when it is known that every coefficient must be less than some value, for instance, a factor of the largest program constant, or there are no cyclical dependent cutpoints that forces to check simultaneously satisfiability of the constraints associated with multiple paths. On the other hand, there is the first criterion that establishes a limit which comes on when the second one fails.

5.3. Tool distribution

CppInv is available for download together with all examples shown in this document in the following URL (last online checking on 2011/06/23):

`http://www.lsi.upc.edu/~albert/cppinv.tar.gz`

6 Conclusions and futher work

To conclude, we summarize the contributions of this thesis and examine possible further work.

6.1. Results of the thesis

We have presented a tool which generates linear loop invariants for programs written in a subset of C++. The tool is fully automatic and only generates invariants that are not linear combination of previous ones which is done using an original non-linear arithmetic encoding.

CppInv makes extensive use of SMT-solvers. In particular, the tool uses the Barcelogic solver for non-linear arithmetic. Our tool has been succesfully applied to a set of programs coming from the literature including consecutive and nested loops and up to 50 lines of code. To illustrate the kind of programs we can handle, some representative ones are listed in the appendix **A**. This examples are taken from the literature, for instance, from [1], [5], [6], [7], [10] and [11].

Finally, let us mention that as a by-product of our tool we can detect unreachable locations in the program.

6.2. Further work

There exist various research lines, some ones related to invariant generation and other ones to its application.

On the one hand, it is expected to design and implement methods for asserting about integer arrays and to extend support for non-linear expressions and invariants.

On the other hand, from the invariance information obtained with the presented approach, new techniques will be developed to prove automatically termination of imperative programs. Our ideas, start from a similar frame than the explained for invariant generation. In particular, it is considered an alike approach to the one presented in [7] but improving the techniques for automatic generation of lexicographic ranking functions. For that, we use ideas to extract lexicographic measures from the analysis of the strongly connected components of the control flow graph built for a iterative system such as the ones described in [8]. Notice that, surprisingly, the achieved progress for termination of imperative programs and termination of rewriting systems, despite the existence of numerous connecting links, it was developed independently without using the knowledge of both areas.

Finally, there is a lot of work to be done on the improvement of the non-linear arithmetic solver. In particular, we would like to study the combination of our current approach based on proving satisfiability with other methods based on proving unsatisfiability, and extend the solver to combine real and integer variables.

7 References

[1] M.A. Colón, S. Sankaranarayanan and H.B. Sipma. “Linear invariant generation using non-linear constraint solving”. In Proceedings of 15th International Conference on Computer Aided Verification, CAV’03. Lecture Notes in Computer Science, vol. 2725, pp. 420–432. Springer, 2003.

[2] P. Cousot and R. Cousot. “Abstract Interpretation: A unfied lattic model for static analysis of programs by construction or approximation of fixpoints.” In ACM Principles of Programming Languages, pp. 238–252, 1977.

[3] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell and A. Rubio. “SAT Modulo Linear Arithmetic for Solving Polynomial Constraints”. Journal of Automated Reasoning, to appear 2011. Springer-Verlag. DOI: 10.1007/s10817-010-9196-8

[4] A. Tarski. “A decision method for elementary algebra and geometry”. Univ. of California Press, Berkeley, 5, 1951.

[5] A.R. Bradley, Z. Manna and H.B. Sipma. “Termination analysis of integer linear loops”. Journal of Concurrency Theory, pp. 488–502. Springer, 2005.

[6] S. Sankaranarayanan, H.B. Sipma and Z. Manna. “Non-linear Loop Invariant Generation using Gröbner Bases”. In Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL’04, pp. 318–329. ACM Press, New York (2004).

[7] A.R. Bradley, Z. Manna and H.B. Sipma. “Linear Ranking with Reachability”. In Proceedings of 17th International Conference on Computer Aided Verification, CAV’05. Lecture Notes in Computer Science, vol. 3576, pp. 491–504. Springer, 2005.

[8] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. “Mechanizing and Improving Dependency Pairs”. Journal of Automated Reasoning, 37(3): 155–203. Springer-Verlag, 2006.

[9] W. Hodges. “Model theory”. Enclyclopedia of mathematics and its applications, vol. 42. Cambridge University Press, Cambridge, MA.

[10] D. Kapur. “Automatically generating loop invariants using quantifier elimination”. In IMACS Intl. Conf. on Applications of Computer Algebra, 2004.

[11] M.A. Colón and H.B. Sipma. “Practical methods for proving program termination”. Journal of Computer Aided Verification, pp. 227–240. Springer, 2002.

[12] P. Cousot and H. Halbwachs. “Automatic discovery of linear restraints among the variables of a program”. In ACM Principles of Programming Languages, pp. 84–97, January 1978.

[13] F. Besson, T. Jensen and J.-P. Talpin. “Polyhedral analysis of synchronous languages”. In Static Analysis Symposium, SAS’99, Lecture Notes in Computer Science 1694, pp. 51–69, 1999.

[14] A. Gupta and A. Rybalchenko. “InvGen: An efficient invariant generator”. Journal of Computer Aided Verification, pp. 634–640. Springer, 2009.

[15] S. Ranise and C. Tinelli. “The SMT-LIB standard: Version 1.2”. Journal of Department of Computer Science, The University of Iowa, Tech. Rep., 2006.

A Applications

This appendix collects some program examples and discusses the invariants generated automically by CppInv.

A.1. Newton-Raphson's algorithm

Consider the next program, taken from [10], for computing the floor of the square root of a natural number N using the Newton-Raphson's method.

```
int main()
{
   int a=0, s=1, t=1, N;
L: while (s <= N)
   {
     a=a+1;
     s=s+t+2;
     t=t+2;
   }
}
```

Figure A.1 Newton-Raphson's method

Given as input to CppInv, it produces the following linear invariants for the cutpoint labeled with L:

$$\varphi_0 : a - t + 1 \leq 0, \quad \varphi_1 : 2a - t + 1 \leq 0, \quad \varphi_2 : -2a + t - 1 \leq 0$$
$$\varphi_3 : a - s + t \leq 0, \quad \varphi_4 : 55a - 10s + 3t - 36 \leq 0$$

From φ_1 and φ_2 we can obtain $2a - t + 1 = 0$ which according to [10] is the strongest linear invariant equality of L.

Futhermore, if we ask CppInv for discovering invariants that also may contain monomials a^2, at and t^2, we obtain the following invariants:

$$\begin{aligned}
\varphi_0 &: a - a^2 + at - s + 1 \leq 0, & \varphi_1 &: -6a + 2a^2 - 2at + 6s - t^2 - 5 \leq 0\\
\varphi_2 &: -a - a^2 + at - s + t \leq 0, & \varphi_3 &: 4a + 6a^2 - 6at + 2s - 3t + t^2 \leq 0\\
\varphi_4 &: -a - a^2 - at - s + t^2 \leq 0, & \varphi_5 &: 6a + 2a^2 - 2at + t - 5 \leq 0\\
\varphi_6 &: -a + a^2 - s + t \leq 0, & \varphi_7 &: a + a^2 + at + s - t^2 \leq 0\\
\varphi_8 &: at + s + t - t^2 - 1 \leq 0
\end{aligned}$$

A.2. LCM-GCD Algorithm

Figure A.2 shows a program, taken from [6], that calculates simultaneously the LCM and GCD of integers $x1$ and $x2$. The selected cutset is $\{L1, L2\}$, although we can force CppInv to include $L0$ if we ask it for avoiding cutset simplification. In order to obtain interesting invariants, monomials $y1y3$, $y2y4$ and $x1x2$ must be contained. In this case, it is only necessary to pass as an argument the monomial $y1y3$ because $y2y4$ and $x1x2$ are added automatically as a requirement for calculating the next value of $y1y3$ through the different paths. As a result, CppInv obtains the invariants $\varphi_0 : x1x2 - y1y3 - y2y4 \leq 0$, $\varphi_1 : -x2 + y2 \leq 0$ and $\varphi_2 : -x1x2 + y1y3 + y2y4 \leq 0$ at $L0$, the same invariants at $L1$, and φ_0, φ_2 and $\varphi_3 : -x2 + y1 \leq 0$ at $L2$.

Taking into account that invariant $\varphi_4 : y1y3 + y2y4 = x1x2$ is a logical consequence of φ_0 and φ_2, and applying the exit condition $y1 = y2$ yields, $y1(y3 + y4) = x1x2$. Assuming that $y1 = GCD(x1, x2)$ and $y3 + y4 = LCM(x1, x2)$, the invariant states that

$$LCM(x1, x2) \cdot GCD(x1, x2) = x1x2$$

Note that correctness cannot be inferred directly since LCM and GCD functions cannot be expressed algebraically.

```
int main()
{
    int x1,x2;
    int y1=x1, y2=x2, y3=x2, y4=0;
L0: while(y1 != y2)
    {
L1:   while (y1>y2) {
        y1=y1-y2;
        y4=y4+y3;
      }
L2:   while (y2>y1) {
        y2=y2-y1;
        y3=y3+y4;
      }
    }
    // y1=GCD(x1,x2) /\ y3+y4=LCM(x1,x2)
}
```

Figure A.2 Simultaneous LCM-GCD algorithm

A.3. Generalized Readers-Writers

Consider the program shown in Figure A.3 modeling a generalization of the readers-writers problem taken from [6] The number of readers and writers are represented by r and w respectively. Initially we assume $k = k0$ tokens to be present. The call to `get_operation` returns a number code between 0 and 3 encoding four possible actions. If the action is to obtain reading access (taking $c1$ tokens) or to read data, `get_operation` checks no writers are present so that we can ensure $w = 0$ holds within the corresponding block of code modeling the action. Similarly, if the action is to obtain writing access (taking $c2$ tokens) or to write data, `get_operation` checks no readers are present so that we can ensure $r = 0$ holds within the corresponding block of code modeling the action. Indicating to CppInv that we are interested about invariants that contains the quadratic monomials $rc1$, $wc2$ and rw apart from single variables, we obtain the invariants $\varphi_0 : rw \leq 0$, $\varphi_1 : -rw \leq 0$, $\varphi_2 : k - k0 + rc1 + wc2 \leq 0$ and $\varphi_3 : -k + k0 - rc1 - wc2 \leq 0$.

Notice that φ_0 and φ_1 implies $\varphi_4 : rw = 0$, which establishes mutual exclusion between the readers and the writers, while φ_2 and φ_3 implies $\varphi_5 : rc1+wc2+k = k0$, which accounts for the tokens during the run of the program.

A.4. Heapsort

Figure A.4 shows program HeapSort taken from [1]. All cycles are cut by the location labeled with `L`.

If we give as input the program to CppInv, the tool iterates three times before it outputs the final set of invariants for `L`. In each iteration, CppInv uses the new found invariants as feedback by following the procedure explained in Section 4.7.

In the first iteration, CppInv generates the invariants $\varphi_0 : -n + r \leq 0$, $\varphi_1 : -l + 1 \leq 0$, $\varphi_2 : 2i - j \leq 0$ $\varphi_3 : -2i + j \leq 0$ and $\varphi_4 : 2l - r \leq 0$. Notice that from φ_2 and φ_3 we can obtain $2i = j$, that φ_1 and φ_4 implies $r \geq 2$, and from this last invariant and φ_0 we can obtain $n \geq 2$. All these invariants match with the obtained ones in [1].

In the second iteration, CppInv finds two new invariants which are not implied from the previous ones: $\varphi_4 : i \leq r$ and $\varphi_5 : 2l \leq j + 1$. From these invariants we can entail the rest of invariants obtained in [1].

The final set of invariants after the last iteration contains φ_2, φ_3, φ_4 and also the new invariants $\varphi_5 : j + l - n - r + 1 \leq 0$ $\varphi_6 : i + j - l - n \leq 0$, $\varphi_7 : -i + j - l + n - r \leq 0$, $\varphi_8 : -j - l + r \leq 0$ and $\varphi_9 : -j - n + 2r - 1 \leq 0$. Notice that φ_0, φ_1 and φ_4 has been removed because they are implied by the rest of invariants. Furthermore, since we have obtained new invariants and the method ensures they are not logical consequences of previous ones, then the conjunction of the final invariants are stronger than the given ones in [1].

A.5. MergeSort

Consider the program MergeSort shown in Figure A.5 taken from [11]. The selected cutpoints are the locations labeled with $L0$, $L1$, $L2$ and $L3$. This example can also be handled by CppInv. However, it takes a long execution time. The main reason is the existence of 3 cyclical dependent cutpoints ($L0$, $L1$ and $L2$) and multiples paths that join them. This produces an SMT problem that involves a great amount of variables.

The tool finds 12 invariants for $L0$, $L1$ and $L3$, and 11 invariants for $L2$. For instance, a found invariant for location $L0$ is $j + 1 = i + m + q + r$. Futhermore, before obtaining the final set of invariants for each cutpoint, CppInv generates more than 450 intermediate invariants.

A.5.MergeSort

```
int main()
{
  int r=0, w=0, k0, k=k0, c1, c2;
  while (true)
  {
    int OP=get_operation(r,w); // check no readers or no writers are present
                               // according to the requested operation
    assert(0 <= OP and OP <= 3);
    // OP=0 --> Obtain reading access; OP=1 --> Read data
    // OP=2 --> Obtain writing access; OP=3 --> Write data
    if (OP==0)
    {
      assert(w==0);
      r=r+1;
      k=k-c1;
    }
    else if (OP==1)
    {
      assert(w==0);
      r=r-1;
      k=k+c1;
    }
    else if (OP==2)
    {
      assert(r==0);
      w=w+1;
      k=k-c2;
    }
    else if (OP==3)
    {
      assert(r==0);
      w=w-1;
      k=k+c2;
    }
  }
}
```

Figure A.3 Generalized readers-writers

```
int main()
{
  int n, i, j, k;
  int T[n+1];
  assert(n>=2);
  int r=n, l=n/2+1;
  if (l>=2) {
    k=T[l];
    --l;
  }
  else {
    k=T[r];
    T[r]=T[1];
    --r;
  }
  while (r>=2) {
    i=l;
    j=2*l;
L:  while (j<=r) {
      if (j<=r-1 && T[j]<T[j+1]) ++j;
      if (k>=T[j]) break;
      T[i]=T[j];
      i=j;
      j*=2;
    }
    T[i]=k;
    if (l>=2) {
      k=T[l];
      --l;
    }
    else {
      k=T[r];
      T[r]=T[1];
      --r;
    }
    T[1]=k;
  }
}
```

Figure A.4 HeapSort

A.5.MergeSort

```
int main() {
  int n;
  assert(n>0);
  int a[2*n];
  int i,j,k,l,t,h,m,p,q,r;
  bool up=true;
  p=1;
  do {
    h=1; m=n;
    if (up) {
      i=1; j=n; k=n+1; l=2*n;
    }
    else {
      k=1; l=n; i=n+1; j=2*n;
    }
    do {
      if (m >= p) q=p; else q=m;
      m-=q;
      if (m >= p) r=p; else r=m;
      m-=r;
L0:   while (q>0 and r>0) {
        if(a[i]<a[j]){
          a[k]=a[i];
          k+=h; ++i; --q;
       }
       else {
          a[k]=a[j]; k+=h; --j; --r;
       }
      }
L1:   while (r>0) {
       a[k]=a[j]; k+=h; --j; --r;
      }
L2:   while (q>0) {
         a[k]=a[i];
         k+=h; ++i; --q;
      }
      h=-h; t=k; k=l; l=t;
    } while (m<=0);
    up=!up; p*=2;
  } while (p>=n);

  if (!up) {
    i=1;
L3: while (i<=n) {
      a[i]=a[i+n]; ++i;
    }
  }
}
```

Figure A.5 MergeSort