Automatic Invariant Generation

Nishant Totla

***Abstract*—Program invariants are statements or assertions that are widely used in program analysis and proving correctness of programs. A program invariant can be defined for every program location. Invariants provide properties that hold for every valid program execution, and are crucial in program verification. In this survey, we shall describe some techniques that have been used in the past to generate invariants by analyzing programs.**

I. Introduction

Traditional program verification relies on static analysis to prove the absence of errors in the program. By a straightforward reduction to the halting problem it is possible to prove that (for any Turing complete language) finding all possible run-time errors in an arbitrary program (or more generally any kind of violation of a specification on the final result of a program) is undecidable: there is no mechanical method that can always answer truthfully whether a given program may or may not exhibit runtime errors. This result dates from the works of Church, Gödel and Turing in the 1930s. As with many undecidable questions, one can still attempt to give useful approximate solutions.

Program invariants are logical assertions that are preserved as a program executes. The problem of inferring these assertions directly relates to searching for errors, because invariants characterize exactly those states that are legal at a certain program location. Hoare logic [1] introduces a convenient notation to describe how the execution of a piece of code affects the state of the computation: $\{P\}C\{Q\}$, where C is a piece of code, and P and Q are the preconditions and postconditions respectively. This means that if execution starts in a state satisfying P, then if it terminates, it terminates in a state satisfying Q. Hoare logic provides rules of composition that can be used to prove the correctness of a given Hoare triple $\{P\}C\{Q\}$. But loops present a problem. When C is a loop construct, the composition rules demand a loop invariant.

A loop invariant is a predicate that holds on entry into the loop, and is preserved in each iteration of the loop. One technique for reasoning about loops is to unroll them, until one proves there cannot be another iteration. However, this approach does not scale to programs where loops have many iterations. To combat this problem, we need techniques to infer loop invariants. Computing weakest preconditions for loops also requires loop invariants.

Programmers may or may not know logic, and hence one cannot expect them to provide program invariants at every step. Thus, it makes sense to have automated techniques to generate these invariants.

As an example of the kind of assertions we wish to infer, consider Listing 1, which is a non-trivial sorting program. I will describe techniques that generate invariants like the following (by line number)

```
1 // Sort array K
2 int B,J,T;
3 B = N;
4 while(B>=1){
5   J=1; T=0;
6   while(J<=(B-1)){
7     if(K[J]>K[J+1]){
8       EXCHANGE(J,J+1);
9       T=J;
10    }
11    J=J+1;
12  }
13  if(T==0) return;
14  B=T;
15 }
```

Listing 1. Sample Bubblesort program from [2].

$$
\begin{array}{rl}
3 & B = N \\
4 & 1 \leq B \leq N \\
5 & 1 \leq B \leq N, J = 1, T = 0 \\
6{,}7{,}8 & B \leq N, T \geq 0, T + 1 \leq J, J + 1 \leq B \\
9 & B \leq N, J \geq 1, J + 1 \leq B, J = T \\
10 & B \leq N, J + 1 \leq B, J \geq 1, T \geq 0, T \leq J \\
11 & B \leq N, J \leq B, J \geq 2, T \geq 0, T + 1 \leq J \\
12{,}13 & B \leq N, J \leq B, T \geq 0, T + 1 \leq J, B \leq J + 1 \\
14 & J \leq N, T \geq 0, T + 1 \leq J, T = B \\
15 & B \leq N, B \leq 1
\end{array}
$$

These invariants could be used as assertions at corresponding program locations to catch any erroneous executions. Invariants are useful for many applications in program analysis such as runtime safety, reachability, deadlock-freedom, array-bounds analysis, memory safety, pointers analysis, and compiler optimizations.

II. Outline and Preliminaries

The basic techniques surveyed in this report will demonstrate the different ways in which the problem of invariant inference can be tackled. Starting with the least fixed-point computation approach of [3], we explore the propagation-based technique of [2] which uses Abstract Interpretation [4]. Finally, we look at a "bottom-up" approach that generates linear invariants which are correct by construction [5].

We also analyze the pros and cons of all these approaches, and also refer to some more recent and domain-specific approaches. Finally, we look at some implementations that compute program invariants, and the techniques they utilize.

A. Preliminaries: Inductive Invariants

For loops, the notion of inductive invariants is defined as follows. We say that φ is an inductive invariant if it holds at the

start of the loop (initiation) and each loop iteration preserves it (consecution). Inductive invariants are key for loops, because they help in computing the verification conditions for loops.

B. Preliminaries: Assertion Domains

The assertion domain denotes the class of assertions which contains the initial assertions, the transition relations, and the invariants. Some commonly used assertion domains are

- Linear equality over reals ($2i + j - 3 = 0$)
- Linear inequalities over reals ($2i + 3j - 2 \leq 0$)
- Integer Arithmetic ($\exists k(2j = i \wedge i = j + k)$)
- Multiplication over reals ($\exists a, b(ai^3 + bi = j \vee i = j^2)$)

Usually, invariants are computed for fixed assertion domains. Fixing an assertion domain helps us by providing a known set of procedures (for the corresponding assertion domain) that can be directly used in invariant generation. Additionally, extraneous operations, that might potentially be complex can be skipped to restrict the assertion domain.

We now describe some interesting theoretical techniques to compute invariants.

III. Fixed point computations

The techniques in this section were first presented in [3], in light of the fixpoint approach in the static analysis of programs. This paper gives lattice theoretic foundations and deductive semantics of programs, using which, we can, in theory, infer optimal invariants for total correctness proofs. Let us first look at some mathematical preliminaries before demonstrating this method on a toy example.

A. Lattice Theory

Consider a complete lattice $(L, \preceq, \vee, \wedge, \top, \bot)$. The following facts will be needed for this section

- A function $f : L \to L$ is called isotone/monotone if it is order-preserving.
- Consider a chain of elements of L: $x_0 \preceq x_1 \preceq x_2 \preceq \cdots$. Then the limit of this chain is defined as $\vee_{i=0}^{\infty} x_i$.
- $f : L \to L$ is defined to be continuous if for any sequence $\{x_i\}$ that converges to x, we have $f(x) = lim\{f(x_i)\}$.
- Continuous functions can be shown to be monotone.

In our analysis, given the complete lattice L, we need to consider the lattice L^n, which can be shown to be a complete lattice (the lattice operations are simply projected on each component). Transforming L to L^n preserves monotonicity of functions.

We now consider the fixed-point equation $x = f(x)$. More elaborately, it is written as

$$\begin{aligned} x_1 &= f(x_1, \cdots, x_n) \\ x_2 &= f(x_1, \cdots, x_n) \\ &\vdots \\ x_n &= f(x_1, \cdots, x_n) \end{aligned} \tag{1}$$

Any solution that satisfies $x = f(x)$ is a fixed-point of f.

Definition 1. *x^* is a least fixed-point of f if $x^*=f(x^*)$ and for every x such that $x=f(x)$, $x^* \preceq x$.*

We now describe Tarski's Theorem, that guarantees the existence of a least fixed-point.

Theorem 1 (Tarski)**.** *Any monotone map f of a complete lattice L^n to itself has a least fixed-point defined by $lfp(f)=\wedge\{x \in L^n | f(x) \preceq x\}$*

Note that Tarski's theorem is non-constructive. But it can be shown that if f is also continuous, then the least fixed-point can be approximated as the limit of a sequence of successive approximations: $X^0 = f(\bot), X^1 = f(X^0), \cdots, X^k = f(X^{k-1}), \cdots$, that is, $lfp(f) = lim_{k\to\infty} f^k(\bot)$[1].

[3] generalizes this by showing that any chaotic iteration also converges[2].

B. Deductive Semantics of Programs

We define assertions for every program point i.

Definition 2. *$P_i(x, \overline{x})$ is a logical first order predicate over the set x or program variables at line i and the set $\overline{x}$ of initial values of these program variables.*

Let L be the set of all such predicates $P(x, \overline{x})$. Then this set forms a complete lattice $(L, \preceq, \vee, \wedge, \top, \bot)$ given by $(L, \Rightarrow, or, and, true, false)$.

C. System of Logical Forward Equations

We use the Hoare triple $\{P(x, \overline{x})\}S\{Q(x, \overline{x})\}$ to indicate that Q is the strongest post-condition of P for the execution of S. We can now define the semantics for various program statements.

1) Initialization: At program entry point j, $P_j(x, \overline{x}^j) = \{(x_i = \overline{x}_i^j), i = 1, \cdots, m\}$

2) Assignment Statements: $\{P(x, \overline{x})\}x_i := E(x)\{\exists v \cdot P_{x_i}^v \wedge x_i = E_{x_i}^v\}$
where $P_{x_i}^v = P(x_1, \cdots, x_{i-1}, v, x_{i+1}, \cdots, x_m, \overline{x}_1, \cdots, \overline{x}_m)$
and $E_{x_i}^v = E(x_1, \cdots, x_{i-1}, v, x_{i+1}, \cdots, x_m)$
This rule must be adjusted to handle failed assignments.

3) Conditionals: $\{P(x, \overline{x})\}$ `if` $Q(x)$ `then` $\{P(x, \overline{x}) \wedge Q(x)\} \cdots$ `else` $\{P(x, \overline{x}) \wedge \neg Q(x)\} \cdots$

4) Goto statements and labels: At program location L, the assertion would be $\underset{i \in pred(L)}{\vee} P_i(x, \overline{x})$, where $pred(L)$ is the set of program points going to L sequentially or by jumps.

With these definitions, we illustrate this technique on the toy example in Listing 2. This program has four locations (one before each line number). Hence there will be four assertions P_i for $i = 1, 2, 3, 4$. P_i is the assertion that must hold at the program location right after line $i - 1$.

Using the logical forward equations from Section III-C, we can set up relations between these assertions as follows

[1] This is Jacobi's method of successive approximations.

[2] Chaotic iterations indicate updates where any one or more of the components of the tuple $(x_0, \cdots, x_n)$ can be updated in one step in the successive approximations method, as long as no component is indefinitely ignored.

```
while(x>=y){
  x=x-y;
}
//x and y are integers here
```

Listing 2. Sample program from [3].

$$
\begin{aligned}
P_1(x,y,\overline{x},\overline{y}) &= (x=\overline{x}) \wedge (y=\overline{y}) \\
P_2(x,y,\overline{x},\overline{y}) &= (P_1(x,y,\overline{x},\overline{y}) \vee P_3(x,y,\overline{x},\overline{y})) \wedge (x \geq y) \\
P_3(x,y,\overline{x},\overline{y}) &= \exists v \in \mathbb{Z} \cdot (P_2(x,y,\overline{x},\overline{y}) \wedge (x = v - y)) \\
P_4(x,y,\overline{x},\overline{y}) &= (P_1(x,y,\overline{x},\overline{y}) \vee P_3(x,y,\overline{x},\overline{y})) \wedge (x < y)
\end{aligned} \tag{2}
$$

After this point, for the sake of brevity, we use P_i to mean $P_i(x,y,\overline{x},\overline{y})$. By Tarski's theorem, this system of equations (of the form $P = f(P)$) has a least solution P^{opt}, such that for any other solution $P*$, we have $P^{opt} \Rightarrow P*$. This highlights the importance of the least fixed-point – The least fixed-point gives the strongest invariants. Hence we call them optimal invariants. To compute P^{opt}, we resort to chaotic iterations, starting at the $\perp$ of the lattice, namely $false$. We update P_i in a cyclic manner[3] using values of the previous iteration, and the update rules in Equation 2. Thereby, the successive update steps look as follows

Initialization:

$$
\begin{aligned}
P_1^0 &= false \\
P_2^0 &= false \\
P_3^0 &= false \\
P_4^0 &= false
\end{aligned} \tag{3}
$$

Step 1:

$$
\begin{aligned}
P_1^1 &= (x=\overline{x}) \wedge (y=\overline{y}) \\
P_2^1 &= (\overline{x} \geq \overline{y}) \wedge (x=\overline{x}) \wedge (y=\overline{y}) \\
P_3^1 &= (\overline{x} \geq \overline{y}) \wedge (x=\overline{x}-\overline{y}) \wedge (y=\overline{y}) \\
P_4^1 &= (\overline{x} < \overline{y}) \wedge (x=\overline{x}) \wedge (y=\overline{y})
\end{aligned} \tag{4}
$$

Step 2:

$$
\begin{aligned}
P_1^2 &= (x=\overline{x}) \wedge (y=\overline{y}) \\
P_2^2 &= [(\overline{x} \geq \overline{y}) \wedge (x=\overline{x}) \wedge (y=\overline{y})] \vee \\
&\quad [(\overline{x} \geq \overline{y}) \wedge (\overline{x} \geq 2\overline{y}) \wedge (x=\overline{x}-\overline{y}) \wedge (y=\overline{y})] \\
P_3^2 &= [(\overline{x} \geq \overline{y}) \wedge (x=\overline{x}-\overline{y}) \wedge (y=\overline{y})] \vee \\
&\quad [(\overline{x} \geq \overline{y}) \wedge (\overline{x} \geq 2\overline{y}) \wedge (x=\overline{x}-2\overline{y}) \wedge (y=\overline{y})] \\
P_4^2 &= [(\overline{x} < \overline{y}) \wedge (x=\overline{x}) \wedge (y=\overline{y})] \vee \\
&\quad [(\overline{x} \geq \overline{y}) \wedge (\overline{x} < 2\overline{y}) \wedge (x=\overline{x}-\overline{y}) \wedge (y=\overline{y})]
\end{aligned} \tag{5}
$$

From computing the first few terms, the general term P^i of the sequence is discovered.

[3]For clarity, updates take place in the order $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots$

Step i:

$$
\begin{aligned}
P_1^i &= (x=\overline{x}) \wedge (y=\overline{y}) \\
P_2^i &= \vee_{j=1}^{i} \wedge_{k=1}^{j} [(\overline{x} \geq k\overline{y}) \wedge (x=\overline{x}-(j-1)\overline{y}) \wedge (y=\overline{y})] \\
P_3^i &= \vee_{j=1}^{i} \wedge_{k=1}^{j} [(\overline{x} \geq k\overline{y}) \wedge (x=\overline{x}-j\overline{y}) \wedge (y=\overline{y})] \\
P_4^i &= \vee_{j=1}^{i} \wedge_{k=1}^{j-1} [(\overline{x} \geq k\overline{y}) \wedge (x<j\overline{y}) \wedge \\
&\quad (x=\overline{x}-(j-1)\overline{y}) \wedge (y=\overline{y})]
\end{aligned} \tag{6}
$$

The correctness of P_i is established by mathematical induction on i. The optimal invariants are obtained by computing the limit, $P^{opt} = \lim_{i \to \infty} P_i$. This gives us

$$
\begin{aligned}
P_1^{opt} &= (x=\overline{x}) \wedge (y=\overline{y}) \\
P_2^{opt} &= \exists j \geq 1 \; \forall k \in [1,j][(\overline{x} \geq k\overline{y}) \wedge (x=\overline{x}-(j-1)\overline{y}) \wedge \\
&\quad (y=\overline{y})] \\
P_3^{opt} &= \exists j \geq 1 \; \forall k \in [1,j][(\overline{x} \geq k\overline{y}) \wedge (x=\overline{x}-j\overline{y}) \wedge (y=\overline{y})] \\
P_4^{opt} &= \exists j \geq 1 \; \forall k \in [1,j-1][(\overline{x} \geq k\overline{y}) \wedge (x<j\overline{y}) \wedge \\
&\quad (x=\overline{x}-(j-1)\overline{y}) \wedge (y=\overline{y})]
\end{aligned} \tag{7}
$$

These are optimal invariants, and exactly characterize the allowed program states. These could be used in proofs of total correctness of the program, or for other applications.

[3] goes on to show that symbolic execution [6] essentially solves semantic equations associated with the program, and the construction of the symbolic execution tree corresponds to the chaotic successive approximations method. Thus symbolic execution can be used to discover optimal invariants, provided that the limit can be computed. Either induction principles or difference equations could be used for this purpose.

[3] also provides techniques to compute approximate invariants by using a system of inequations as a substitute for the exact system of equations.

IV. Invariant Generation using Abstract Interpretation

The invariant generation procedure in this section is based on the seminal papers [2][7], which generates linear relationships among program variables. This is one of the most popular techniques, and is based on Abstract Interpretation [4]. While restricting the assertion domain to linear restraints will not allow us reason about all program properties, but it is interesting to note that linear invariants, coupled with analysis using symbolic execution and forward equations can prove a lot of non-trivial properties (alternatively, find bugs).

This technique starts with an initial region and iterates over sets of states. The current iterate is expanded with all states that are reachable in one step from the current. Usually, in such cases, convergence is tricky, because the set of states is infinite, as are the assertions that characterize these states. In this case, convergence is enforced by a heuristic *Widening* technique, that might lead to non-optimal invariants. Hence, the overall performance depends on how well widening is done.

A. Approximate Analysis of Program Properties

A sequential program can be represented by a connected finite graph. It has one entry node and also nodes for assignments, conditionals, junctions and exit. Junction nodes represent nodes where multiple program paths merge, and contain no computations. We also assume a subset of our programming languages in which evaluating $E(x)$ for the assignment $y := E(x)$ has no side-effects. Each edge i connecting two nodes is assigned a predicate $P_i(v_1, v_2, \cdots, v_n)$ where the $v_i s$ are program variables. Such predicates denote relations between program variables, and may be incomplete[4]. The entry node has a dangling incoming edge that asserts the initial conditions.

Given assertions at one node, assertions at successor nodes can be computed using transition relations which might depend on the content of the node. This setup established a system of equations similar to [3]. The entry assertion is propagated along all program paths, and we wish to find a fixed-point.

B. Abstract Values

Let V_c denote the set of concrete values that program variables can take (for example $\mathbb{Z}$). We define the set of abstract values V_a. An abstract value denotes a set of concrete values, or properties of such a set, satisfying a number of dynamic conditions. For example, if $V_c = \mathbb{Z}$, we can take V_a to be the set of all intervals. An abstraction function defines the correspondence between V_c and V_a. Ordering, and operations like union are defined for abstract values.

The interesting case in approximate analysis is that of loop junction nodes. For the abstract evaluation of loops, the problem of computing fixed-points of strictly increasing infinite chains arises. We wish to do this in a finite number of steps. For this purpose, an operation called widening has been defined [7], denoted by $\bigtriangledown$. Technically, $\bigtriangledown$ is defined on $V_a \times V_a$. Details can be found in Section 2 of [7]. Specifically for loop junction nodes, let $P_{i_1 j}, P_{i_2 j}, \cdots, P_{i_m j}$ be the assertions associated with the input edges of a junction node at step j. Then the assertion Q_j associated with the output edge of this node is given by $Q_j = f(P_{i_1 j}, \cdots, P_{i_m j})$. For a loop junction node, we have $Q_j = Q_{j-1} \bigtriangledown_j f(P_{i_1 j}, \cdots, P_{i_m j})$, where the widening operator $\bigtriangledown_j$ performed at step j satisfies $Q \cup P \Rightarrow Q \bigtriangledown_j P$, and the chain $Q_0 = P_0, Q_1 = Q_0 \bigtriangledown_1 P_1, \cdots, Q_j = Q_{j-1} \bigtriangledown_j P_j, \cdots$ is not an infinite strictly increasing chain.

C. Formal Representation of Linear Restraints

Since we are trying to infer linear restraints among the program variables, the set of allowed states at a program location can be represented as a polyhedron in $\mathbb{R}^n$ (where n is the number of variables in the program). There are multiple ways to represent a polyhedron, and as we shall see, different representations make different computations easy, so it is essential to look at the representations we will need, and how to convert between them.

[4]In the sense that they are only partial constraints.

1) Linear System of a convex polyhedron: This representation uses a set of linear inequalities to characterize convex polyhedra. We consider only closed polyhedra for technical reasons. Any solution to the system of inequalities is a point that belongs to the polyhedron.

2) Frame of a convex polyhedron: This representation uses three sets to characterize closed convex polyhedra: (S, R, D). $S = \{s_1, \cdots, s_\sigma\}$ is the set of vertices of the polyhedron. $R = \{r_1, \cdots, r_\rho\}$ is the set of rays in the polyhedron. $D = \{d_1, \cdots, d_\delta\}$ is the set of lines in the polyhedron. To check if a point $x \in P$, where P is a closed convex polyhedra, the following equivalence holds:
$(x \in P) \Leftrightarrow (x = a + b + c)$ where a is a convex combination of the vertices S, b is a conic combination of the rays R, and c is a linear combination of the lines D.

3) Conversions between the two representations: We shall skip these conversion algorithms in the interest of space in this paper, but they have been described in detail in [2]. In addition, algorithms to simplify a given frame or a given linear system have also been described.

This method of inferring restraints involves propagating convex polyhedra along program execution, and both representations (linear system and frame) are maintained while doing the computations. This is because in practice, both representations are individually, or together important for efficient computations, and inter-conversion between representations is expensive.

D. Transformation of Linear Assertions (polyhedra) by Elementary Language Constructs

The abstract values set V_c in our case is the set of closed convex polyhedra. We now specify guidelines on how individual languages constructs propagate polyhedra, by analyzing various cases. Note that there is guessing and heuristics here, and they turn out to be sufficient to infer many non-trivial invariants. Although more fine-grained analysis might be needed for better results.

1) Program Entry: Depending on what constraints are known between input variables, the starting polyhedron will be initialized. In case there are no constraints, we essentially start with the entire space $\mathbb{R}^n$.

2) Assignment: Suppose an assignment statement transforms the polyhedron represented by (A, B, S, R, D) (both linear system and frame included) to the polyhedron (A', B', S', R', D').

Assignment of a non-linear expression: In the coarsest analyses, we assume the non-linear assignment to be a random assignment, about which nothing is known. If the assignment is $x^{l_0} = E(x)$, then the update amounts to eliminating the variable x^{l_0} from the system by projection (See [2] for details and an example). It is possible to do better analysis by extra knowledge on the form of the expression E, but that depends on the operators used.

Assignment of a linear expression: This assignment is of the form $x^{l_0} = \sum_{i=1}^{n} (a_i x^i) + b$. This results in adding an extra constraint to the linear system. The frame is updated accordingly.

3) Conditional Statements: If P denotes the input polyhedron for the conditional (with boolean condition C), and P_t and P_f denote the polyhedra at the start of the $true$ and $false$ branches respectively, then we know that informally (with some abuse of notation), $P_t = P \wedge C$ and that $P_f = P \wedge \neg C$. Depending on the structure of C, we have the following cases
Non-linear tests: If C is a non-linear condition, we ignore the test and set $P_t = P_f = P$. While this is sound, it might miss a lot of information. Specific analysis is required for special non-linear tests.
Linear equality tests: The linear equality condition C characterizes a hyperplane H. If P is included in H, then $P_t = P \cap H$ and $P_f = \phi$. In case P is not included in H, then we assign $P_t = P \cap H$ and $P_f = P$. Note that $P_f = P$ is an approximation to $P_f = P \wedge \neg C$, because the latter in general is not a closed convex polyhedron.
Linear inequality tests: This is easy. The only thing to be careful about is that C or $\neg C$ might not be closed polyhedra, in which case their closure must be taken while intersecting with P.

4) Simple Junction Nodes: This corresponds to the program nodes where the if and else branches merge in. For such a junction node, the resulting polyhedron should simply be a disjunction of all polyhedra propagated by the incoming edges. But the problem is that a disjunction of closed convex polyhedra might not even be convex, so we take the convex hull of the disjunction. Note that the frame representation really helps in making this computation less expensive, as opposed to the linear system.

5) Loop Junction Nodes: Consider a loop junction node, with incoming edges carrying polyhedra $P_1, \cdots, P_m$ and the outgoing edge polyhedron be P. Then the corresponding transformation that we define is $P = P \bigtriangledown convexhull(P_1, \cdots, P_m)$. In this case, we define the widening operator as follows - $Q_1 \bigtriangledown Q_2$ is the convex polyhedron consisting in the linear restraints of Q_1 verified by every element of the frame of Q_2. As an example, consider

$$\begin{aligned} Q_1 &= \{-x_1 + 2x_2 \leq -2, x_1 + 2x_2 \leq 6, x_2 \geq 0\} \\ Q_2 &= \{-x_1 + 2x_2 \leq -2, x_1 + 2x_2 \leq 10, x_2 \geq 0\} \\ Q_1 \bigtriangledown Q_2 &= \{-x_1 + 2x_2 \leq -2, x_2 \geq 0\} \end{aligned} \tag{8}$$

Efficient computation in this part requires both the linear system and frame representations. The definition of the widening operation must be a balance between forcing the convergence of the global analysis of the program and discovering as much information as possible about the program. Hence, usually it is wise not to perform widening operations at loop junction nodes before gathering information along the program cycles containing that loop junction node.

E. Global Analysis (example)

We illustrate this method on the program in Listing 3. The condition C in the if statement is non-linear, so it will be ignored in the analysis.

```
1 int I,J;
2 //P_0
3 I=2; J=0;
4 //P_1
5 while(true){
6 //P_2
7   if(C){
8 //P_3
9     I=I+4;
10 //P_4
11   }
12   else{
13 //P_5
14     J=J+1;
15     I=I+2;
16 //P_6
17   }
18 //P_7
19 }
20 //Comments denote the assertions that hold at
     the indicated locations
```

Listing 3. Sample program from [2].

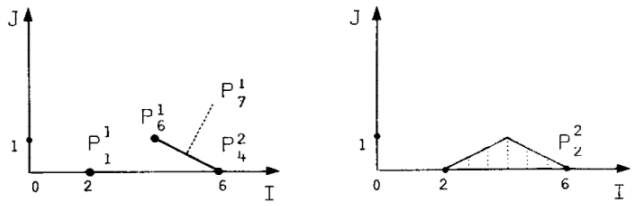

Fig. 1.

Each assertion $P_i^0, i = 0, \cdots, 7$ is initially the empty polyhedron ϕ and the input assertion is propagated through the program graph[5].
$P_0^1 = \mathbb{R}^2$
$P_1^1 = \{I = 2, J = 0\}$
$P_2^1 = convexhull(P_1^1, P_7^0) = convexhull(P_1^1, \phi) = P_1^1$
$P_3^1 = P_5^1 = P_2^1$
$P_4^1 = \{I = 6, J = 0\}$
$P_6^1 = \{I = 4, J = 1\}$
$P_7^1 = convexhull(P_4^1, P_6^1) = \{I + 2J = 6, 4 \leq I \leq 6\}$

$P_2^2 = convexhull(P_1^1, P_7^1) = \{2J + 2 \leq I, I + 2J \leq 6, 0 \leq J\}$
$P_3^2 = P_5^2 = P_2^2$
$P_4^2 = assign(P_3^2, I := I + 4) = \{2J + 6 \leq I, I + 2J \leq 10, 1 \leq J\}$
$P_6^2 = assign(assign(P_5^2, J := J + 1), I := I + 2) = \{2J + 2 \leq I, I + 2J \leq 10, 1 \leq J\}$
$P_7^2 = convexhull(P_4^2, P_6^2) = \{2J + 2 \leq I, 6 \leq I + 2J \leq 10, 0 \leq J\}$

When the loop body has been analyzed, a widening operation takes place at the loop junction node (line 5 in Listing 3):

[5]For simplicity, we only include the linear system representation of the polyhedra here, and skip the frame representation altogether. But the frame is maintained, and actively used.

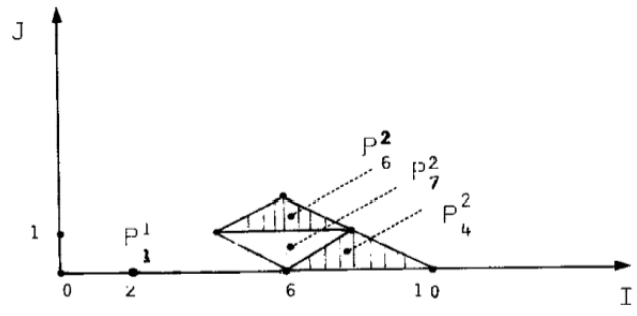

Fig. 2.

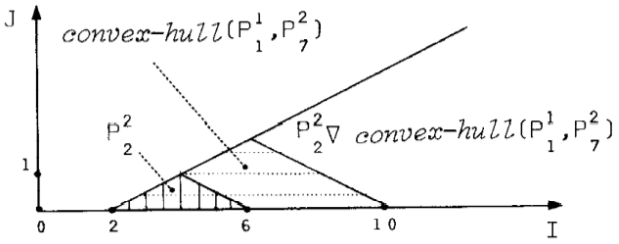

Fig. 3. The widening operation

$P_2^3 = P_2^2 \bigtriangledown convexhull(P_1^1, P_7^2)$.
We have $P_2^2 = \{2J + 2 \leq I, I + 2J \leq 6, 0 \leq J\}$ and $convexhull(P_1^1, P_7^2) = \{2J + 2 \leq I, I + 2J \leq 10, 0 \leq J\}$. At this point, to complete the widening operation, we eliminate the constraint $I + 2J \leq 6$ from P_2^2 as not all points of $convexhull(P_1^1, P_7^2)$ satisfy it. Figure 3 shows this clearly.

We continue propagation again
$P_2^3 = \{2J + 2 \leq I, 0 \leq J\}$
$P_3^3 = P_5^3 = P_2^3$
$P_4^3 = assign(P_3^3, I := I + 4) = \{2J + 6 \leq I, 0 \leq J\}$
$P_6^3 = assign(assign(P_5^3, J := J + 1), I := I + 2) = \{2J + 2 \leq I, 1 \leq J\}$
$P_7^3 = convexhull(P_4^3, P_6^3) = \{2J + 2 \leq I, 6 \leq I + 2J, 0 \leq J\}$

At this stage, $convexhull(P_1^1, P_7^3)$ is included in P_2^3, so the program analysis has converged. We notice that the final result shows up linear restraints among program variables that are never specified in the program itself, and are non-trivial. We end up with the following invariants after the corresponding line numbers (in Listing 3):

1 no information
3 $I = 2, J = 0$

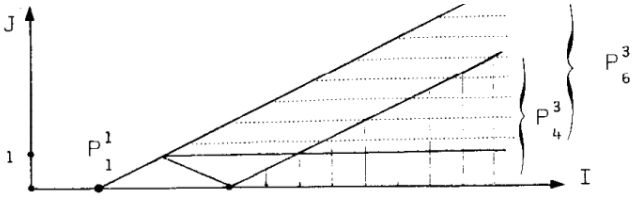

Fig. 4.

5,7,12 $2J + 2 \leq I, J \geq 0$
9 $2J + 6 \leq I, J \geq 0$
15 $2J + 2 \leq I, J \geq 1$
17 $2J + 2 \leq I, 6 \leq I + 2J, J \geq 0$

Also note that it is possible to infer inductive invariants from this information, by collecting all assertions that are true at the beginning of the loop and after every iteration. We also note that this technique is general in the sense that it can work for other assertion domains, provided one is able to reason about the appropriate non-linear structures.

V. Constraint Solving

This technique [5] also solves the problem of generating linear invariants, by reducing the problem to a non-linear constraint solving problem. The method is based on Farkas' Lemma, and synthesizes linear invariants by extracting non-linear constraints on the coefficients of a target invariant from a program. The constraints guarantee that the invariant is inductive. Techniques such as specialized quantifier elimination over the reals are applied for solving the constraints. The highlight of this method is that it is complete, as far as inductive invariants (of the template form) are concerned, and is the first sound and complete method for the same. On some examples, it has been able to infer invariants that the Abstract Interpretation approach of Section IV could not, which is confirmed by the experiments of [5]. In addition, there is no heuristic step (such as widening) involved.

A. Transition Systems

A loop is modeled as a transition system.

Definition 3. *A transition system P : $\langle V, L, l_0, \Theta, \mathcal{T} \rangle$ consists of a set of variables V, a set of location L, an initial location l_0, an initial assertion Θ over the variables V, 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 V' represents next-state variables.*

The conditions for inductive invariance are written in two parts: **initiation** and **consecution**. Suppose ψ denotes an invariant, then,

Definition 4. *(Initiation) This states that the initial conditions must imply that the invariant holds at the start of the loop. Mathematically, $\Theta \models \psi$.*

Definition 5. *(Consecution) This states that every loop iteration must preserve the invariant. Mathematically, $\psi \wedge \rho_\tau \models \psi'$.*

B. Farkas' Lemma

Lemma 1. *Let S be a system of linear inequalities over real-valued variables $x_1, \cdots, x_n$, as in Figure 5, and $\psi = c_1 x_1 + \cdots + c_n x_n + d \leq 0$ be a linear inequality. Then $S \models \psi$ iff*

1) *S is unsatisfiable, or*
2) *S is satisfiable, and c_i, d is a conic combination of coefficients a_{ij}, b_j in S*

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

Fig. 5. System of linear inequalities

$$\begin{array}{c|cccccccc} \lambda_0 & & & & & & -1 & \leq 0 & \\ \lambda_1 & a_{11}x_1 & + & \cdots & + & a_{1n}x_n & + & b_1 & \leq 0 \\ \vdots & \vdots & & & & \vdots & & \vdots & \\ \lambda_m & a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & + & b_m & \leq 0 \\ \hline & c_1x_1 & + & \cdots & + & c_nx_n & + & d & \leq 0 \end{array} \Big\} S \quad \leftarrow \psi$$

Fig. 6.

In the case when S is satisfiable, $S \models \psi$ iff there exist multipliers $\lambda_0, \lambda_1, \cdots, \lambda_m \geq 0$ such that:
$c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \cdots, c_n = \sum_{i=1}^{m} \lambda_i a_{in}, d = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$.
These constraints are represented as in Figure 6.

C. Constraint-solving Approach

We first summarize the method informally, and then demonstrate it on a simple example. Let the program variables be $V = \{x_1, \cdots, x_n\}$. There are three primary steps

1) Fix the template of the invariant. In this case, we are targeting linear invariants, so the general invariant will look like $\psi = c_1x_1 + \cdots + c_nx_n + d \leq 0$ for unknown coefficients $c_1, \cdots, c_n, d$.
2) Encode the conditions for invariance (initiation and consecution) as linear inequalities.
3) We want the constraints to entail the invariant, so we use Farkas' lemma to generate conditions on the unknown coefficients. Solving these constraints gives a set of solutions, all of which are legitimate invariants. We complete the set by discovering additional formulas that are implied by the discovered invariants.

D. Demonstration of the Approach

We take the same program that was used in Section IV-E. Although, this program has to be transformed into a transition system to fit the notation we have just defined. The transition system is shown in Figure 7.

For this example, the target invariant would look like ψ : $c_1i+c_2j+d \leq 0$. We now quickly demonstrate how the method works. The first step is to encode the initiation condition. This

integer i, j **where** $i = 2 \wedge j = 0$
l_0 : **while true do**
l_1 : $i := i + 4$ **or** $(i, j) := (i + 2, j + 1)$

$L = \{l_0, l_1\}$, $V = \{i, j\}$,
Θ : $(i = 2 \wedge j = 0)$, $\mathcal{T} = \{\tau_0, \tau_1, \tau_2\}$,
τ_0 : $\langle l_0, l_1, true \rangle$
τ_1 : $\langle l_1,\ l_0,\ (i' = i + 4 \wedge j' = j) \rangle$
τ_2 : $\langle l_1,\ l_0,\ (i' = i + 2 \wedge j' = j + 1) \rangle$

Fig. 7.

$$\begin{array}{c|cccccc} \lambda_0 & & & & -1 & \leq 0 & \\ \lambda_1 & i & & & -2 & = 0 & \Big\} \Theta \Big\{ \begin{array}{rcrcl} i & - & 2 & \leq & 0 \\ -i & + & 2 & \leq & 0 \end{array} \\ \lambda_2 & & j & & = 0 & & \\ \hline & c_1i & + & c_2j & + \quad d & \leq 0 & \leftarrow \psi \end{array}$$

where $\lambda_0 \geq 0$.

$$\exists\ \lambda_0, \lambda_1, \lambda_2\ [\lambda_1 = c_1\ \wedge\ \lambda_2 = c_2\ \wedge\ \cdots\cdots]$$

Fig. 8. Encoding the initiation condition

$$\begin{array}{c|cccccccccc} \lambda_0 & & & & & & & - & 1 & \leq 0 & \\ \mu_1 & c_1i & + & c_2j & & & & + & d & \leq 0 & \leftarrow \psi \\ \lambda_1 & i & & & - & i' & & + & 4 & = 0 & \Big\} \rho_{\tau_1} \\ \lambda_2 & & & j & & & - \quad j' & & & = 0 & \\ \hline & & & & c_1i' & + & c_2j' & + & d & \leq 0 & \leftarrow \psi' \end{array}$$

where $\lambda_0,\ \mu_1 \geq 0$.

$$\exists\ \lambda_0, \lambda_1, \lambda_2, \mu_1\ [\mu_1 c_1 + \lambda_1 = 0\ \wedge\ \cdots\cdots]$$

Fig. 9. Encoding consecution for τ_1

is shown in Figure 8. We eliminate the λs to get the constraint $2c_1 + d \leq 0$.

Similarly, we must encode consecution for τ_1 and τ_2. The encoding for τ_1 is shown in Figure 9. Eliminating the additional variables gives the constraint $(c_1 \leq 0) \vee (c_1 = 0 \wedge c_2 = 0)$. Similarly encoding consecution for τ_2 gives the constraint $(2c_1 + c_2 \leq 0) \vee (c_1 = 0 \wedge c_2 = 0)$.

Combining all these constraints gives us the final constraint: $(2c_1 + d \leq 0) \wedge (c_1 \leq 0) \wedge (2c_1 + c_2 \leq 0)$. It is interesting to note that all solutions to this set of constraints will be genuine inductive invariants. The basic solutions are shown in Figure 10.

Hence, the inductive invariants we get are $j \geq 0$ and $i - 2j \geq 2$. These two together imply $i \geq 2$, which is also an inductive invariant.

c_1	c_2	d	$c_1i + c_2j + d \leq 0$
0	0	-1	$-1 \leq 0$
0	-1	0	$-j \leq 0$
-1	2	2	$-i + 2j + 2 \leq 0$

Fig. 10. Solutions for the system of constraints

VI. Critical Analysis

In this section, we discuss the pros and cons of the approaches described in the previous sections.

A. Fixed-point Method

This method produces powerful invariants, and is equivalent to symbolic execution, loop unrolling and requires guessing of the general term (or using difference equations or induction principles to infer it). But this approach could be hard to scale, and guessing the general term is not easy. Additionally, since the assertion domain is integer arithmetic, the procedure becomes computationally heavy. Most importantly, it involves computing the limit of infinite increasing chains, which might not always converge. Hence this is not practical at all, and must at best be viewed as a theoretical basis for the abstract interpretation approach.

B. Abstract Interpretation

This method can be implemented across numerous assertion domains such as Linear Equalities, Linear Inequalities and Presburger Arithmetic. It can also scale to larger programs over simple domains. For a long time, this was the most popular approach, and many solvers use it in different forms. The major disadvantage, though, is that it involves heuristic widening. If the widening is too general, the invariants will be weak, and if it is too strict, the procedure might take long to converge.

C. Constraint Solving

The most impressive feature of this approach is that it is sound and complete. It involves no heuristic widening, so nothing affects the quality of invariants - all inductive invariants will be generated, or obtained as consequences. This approach was tested in [5] on some non-trivial programs (including HEAPSORT), and it did as well as (in some cases better than) the Abstract Interpretation approach. The major drawback, though, is that if there are too many constraints, they might become hard to solve. Also, the approach as described in Section V is specific to one domain - linear invariants for linear programs. But it can be extended to more domains.

VII. Recent work and other Specific Techniques

In this section, we look at some other techniques for invariant generation, including recent work.

A. Using Local Invariants

[8] generates local invariants using many rules based on the structural properties of the transition system. This applies to sequential transition systems. The technique the considers networks of transition systems and allows to combine local invariants of the sequential components to obtain local invariants of the global system. Furthermore, a refined strengthening technique is used that allows to avoid the problem of size-increase of the considered predicates which is the main drawback of the usual strengthening technique.

B. Non-linear Loop Invariant Generation

Work on generating non-linear invariants has been little, owing to the mathematical or computational difficulty of the problem. Yet, there are some domain-specific methods that have been published.

1) Transformational Approach: Methods that allow to treat combinations of loops are of interest. [9] presents a set of algorithms and methods that can be applied to characterize over-approximations of the set of reachable states of combinations of self-loops. It presents two families of complementary techniques. The first one identifies a number of basic cases of pair of self-loops for which exact characterization of the reachable states is provided. The second family of techniques is a set of rules based on static analysis that allow to reduce multiple self-loops to independent pairs of self-loops. The results of the analysis of the pairs of self-loops can then be combined to provide an over-approximation of the reachable states of the multiple self-loops. This is the first known work on non-linear invariants.

2) Using Gröbner Bases: Using the theory of ideals over polynomial rings, [10] reduces the problem of non-linear invariant generation to a numerical constraint solving problem. This is a constraint-based approach, similar in principle to the one in Section V, where we start with a template invariant and produce constraints on unknown coefficients.

C. Combining Bottom-up and Top-down Propagation-based Techniques

While the strongest invariant can be defined as the least fixed point of the strongest post-condition of a transition system starting with the set of initial states, this symbolic computation rarely converges. [11] presents a method for invariant generation that relies on the simultaneous construction of least and greatest fixed points, restricted widening and narrowing, and quantifier elimination.

D. Inductive learning techniques

Daikon [12] is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. Dynamic invariant detection runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data. Daikon can detect invariants in C, C++, Java, and Perl programs, and in record-structured data sources.

E. Making widening more precise

In Section IV, we noticed that the quality of invariants depends on how precisely the widening operator $\triangledown$ can be defined. Strangely, only in 2004, some 25 years after the original paper [2] appeared, widening was studied in more detail. [13] presents a framework for the systematic definition of new and precise widening operators for convex polyhedra. The framework is then instantiated so as to obtain a new widening operator that combines several heuristics and uses the standard widening (from [2]) as a last resort so that it is never less precise.

VIII. Survey of Tools

Automated loop invariant generation is still an ongoing research effort. Computing invariants is a difficult computation problem, and there is a long way to go before we have a complete optimal procedure. For the purpose of this survey, we tried running some open-source tools.

A. StInG

The Stanford Invariant Generator implements three techniques for the invariant generation problem. Two of them belong to the more traditional propagation-based techniques, primarily Abstract Interpretation [2]. These are contrasted against the constraint-based technique [5]. This tool works on all the benchmark examples that can be downloaded with the tool itself. Scaling issues were not studied in detail.

B. Daikon

The Daikon tool [12] uses inductive learning to generate likely invariants. It ships with a modified version of an old Valgrind source which would not compile for Linux kernels above 2.6[6]. This tool could not be tested.

C. InvGen

InvGen [14] uses a constraint-based approach to generate invariants for imperative programs. InvGen combines it with static and dynamic analysis techniques to solve constraints efficiently. Invgen is only available in binary format, which unfortunately segfaults instantaneously under Linux. This tool could not be tested either.

IX. Conclusion

Invariant generation is a hard problem, and there is no known technique to generate invariants in arbitrary assertion domains. Most approaches therefore, restrict themselves to generate invariants in a specific domain. Each approach has its strengths and weaknesses. Some of them sacrifice completeness or soundness [12] in order to gain efficiency, and others resort to heavy computations but guarantee completeness [5]. Even though there is a lot of effort towards generating linear invariants, the techniques do not scale much. Future work in this area will primarily focus in the following directions

1) Improving existing techniques. For example, by producing better widening operators, or more optimal constraint solvers.
2) Domain specific invariants. For example, programs that manipulate specific data structures that can be reasoned about.
3) Machine Learning. This seems to be a promising direction, and links to many areas such as test-case generation. While soundness might be an issue, these approaches are bound to be highly efficient.
4) Interactive generators. Since invariant generation involves heuristic steps, and the quality of generated invariants depends upon these heuristics, it might be worthwhile to explore how human input could be used to detect patterns or to strengthen invariants.

[6]The current kernel is 3.5

References

[1] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '78. New York, NY, USA: ACM, 1978, pp. 84–96. [Online]. Available: http://doi.acm.org/10.1145/512760.512770

[3] P. Cousot and R. Cousot, "Automatic synthesis of optimal invariant assertions: Mathematical foundations," in *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. New York, NY, USA: ACM, 1977, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/800228.806926

[4] ——, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[5] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear invariant generation using non-linear constraint solving," in *CAV*, 2003, pp. 420–432.

[6] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[7] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[8] S. Bensalem, Y. Lakhnech, and H. Saïdi, "Powerful techniques for the automatic generation of invariants," in *CAV*, 1996, pp. 323–335.

[9] S. Bensalem, M. Bozga, J.-C. Fernandez, L. Ghirvu, and Y. Lakhnech, "A transformational approach for generating non-linear invariants," in *SAS*, 2000, pp. 58–74.

[10] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using grobner bases," 2004.

[11] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, T. Margaria and W. Yi, Eds., vol. 2031. Genova, Italy: Springer-Verlag, Apr. 2001, pp. 113–127.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[13] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella, "Precise widening operators for convex polyhedra," in *Static Analysis: Proceedings of the 10th International Symposium, volume 2694 of Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 337–354.

[14] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *Computer Aided Verification*. Springer, 2009, pp. 634–640.