A Data Driven Approach for Algebraic Loop Invariants[⋆]

Rahul Sharma[1], Saurabh Gupta[2], Bharath Hariharan[2], Alex Aiken[1], Percy Liang[1], and Aditya V. Nori[3]

[1] Stanford University
{sharmar,aiken,pliang}@cs.stanford.edu
[2] University of California at Berkeley
{sgupta,bharath2}@eecs.berkeley.edu
[3] Microsoft Research India
adityan@microsoft.com

Abstract. We describe a GUESS-AND-CHECK algorithm for computing algebraic equation invariants of the form $\wedge_i f_i(x_1, \ldots, x_n) = 0$, where each f_i is a polynomial over the variables $x_1, \ldots, x_n$ of the program. The "guess" phase is data driven and derives a candidate invariant from data generated from concrete executions of the program. This candidate invariant is subsequently validated in a "check" phase by an off-the-shelf SMT solver. Iterating between the two phases leads to a sound algorithm. Moreover, we are able to prove a bound on the number of decision procedure queries which GUESS-AND-CHECK requires to obtain a sound invariant. We show how GUESS-AND-CHECK can be extended to generate arbitrary boolean combinations of linear equalities as invariants, which enables us to generate expressive invariants to be consumed by tools that cannot handle non-linear arithmetic. We have evaluated our technique on a number of benchmark programs from recent papers on invariant generation. Our results are encouraging – we are able to efficiently compute algebraic invariants in all cases, with only a few tests.

Keywords: Non-linear, loop invariants, SMT.

1 Introduction

The task of generating loop invariants lies at the heart of any program verification technique. A wide variety of techniques have been developed for generating linear invariants, including methods based on abstract interpretation [8,13] and constraint solving [7,11], among others.

Recently, researchers have also applied these techniques to the generation of non-linear loop invariants [23,17,21,22,18]. These techniques discover *algebraic invariants*, that is, invariants of the form

$$\wedge_i f_i(x_1, \ldots, x_n) = 0$$

[⋆] This work was supported by NSF grant CCF-0915766.

where each f_i is a polynomial defined over the variables $x_1, \ldots, x_n$ of the program. Note that algebraic invariants implicitly handle disjunctions: if $f_1 = 0 \vee f_2 = 0$ is an invariant then $f_1 = 0 \vee f_2 = 0 \Leftrightarrow f_1 f_2 = 0$. Thus, algebraic invariants are as expressive as arbitrary boolean combinations of algebraic equations.

Most previous techniques for algebraic loop invariants are based on Gröbner bases computations, which cause a considerable slowdown [4]. Therefore, there has been recent interest in techniques for generating algebraic invariants that do not use Gröbner bases [4,18] (see Section 7). In this paper, we address the problem of invariant generation from a data driven perspective. In particular, we use techniques from linear algebra to analyze data generated from executions of a program in order to efficiently "guess" a candidate invariant. This phase can leverage test suites of programs for data generation. This guessed invariant is subsequently checked for validity via a decision procedure. Our algorithm GUESS-AND-CHECK for generating algebraic invariants calls these guess and check phases iteratively until it finds the desired invariant. Failure to prove that a candidate is an invariant results in counterexamples or more data that are used to refine the guess in the next iteration. Furthermore, we are also able to prove a bound on the number of iterations of GUESS-AND-CHECK.

Our guess and check data driven approach for computing invariants has a number of advantages:

– Checking whether the candidate invariant is an invariant is done via a decision procedure. Our belief is that using a decision procedure to check the validity of a candidate invariant can be much more efficient than using it to infer an actual invariant.
– Since the guess phase operates over data, its complexity is largely independent of the complexity or size of the program (the amount of data depends on the number of variables in scope). This is in contrast to approaches based on static analysis, and therefore it is at least plausible that a data driven approach may work well even in situations that are difficult for static analysis. Moreover, the guess step just involves basic matrix manipulations, for which very efficient implementations exist.

There are major drawbacks, both theoretical and practical, with most previous techniques for algebraic invariants. First, these techniques either restrict predicates on branches to either equalities or dis-equalities [6,17], or cannot handle nested loops [15,22], or interpret program variables as real numbers [23,4,21]. It is well known that the semantics of a program assuming integer variables, in the presence of division and modulo operators, is not over-approximated by the semantics of the program assuming real variables. Therefore, these approaches may not produce correct invariants in cases where the program variables are actually integers. Our technique does not suffer from these drawbacks: our check phase can consume a rich syntax and answer queries over both integers and reals (see Section 4.2). Moreover, since these techniques can find algebraic invariants, they can find non-linear invariants representing boolean combinations of linear equalities. If a loop has the invariant $y = x \vee y = -x$ then these techniques can find the invariant $x^2 = y^2$ that is semantically equivalent to the linear invariant:

$$x = y \vee x = -y \Leftrightarrow (x + y)(x - y) = 0 \Leftrightarrow x^2 = y^2$$

But if the invariant is to be consumed by a verification tool that works over linear arithmetic (as most tools do), then $x^2 = y^2$ is not useful. A simple extension to our technique allows us to extract an equivalent (disjunctive) linear invariant from an algebraic invariant when such a linear invariant exists. This extension is possible as our technique is data driven (see Section 5.1).

It is also interesting to note that our algorithm is an iterative refinement procedure similar to the counterexample-guided abstraction refinement (CEGAR) [5] technique used in software model checking. In CEGAR, we start with an over-approximation of program behaviors and perform iterative refinement until we have either found a proof of correctness or a bug. Guess-and-Check is dual to CEGAR – we start with an under-approximation of program behaviors and add more behaviors until we are done. Most techniques for invariant discovery using CEGAR-like techniques have no termination guarantees. Since we focus on the language of polynomial equalities for invariants, we are able to give a termination guarantee for our technique.

Our main contribution is a new sound data driven algorithm for computing algebraic invariants. Specifically:

- We provide a data driven algorithm for generation of invariants restricted to conjunctions of algebraic equations. We observe that a known algorithm [18] is a suitable fit for our guess step. We formally prove that this algorithm computes an under-approximation of the algebraic loop invariant. That is, if G is the guess or candidate invariant, and I is an invariant then $G \Rightarrow I$. This guess will contain all algebraic equations constituting the invariants and possibly more spurious equations.
- We augment our guessing procedure with a decision procedure to obtain a sound algorithm. If the decision procedure successfully answers the queries made, then the output is an invariant and we do generate all valid invariants up to a given degree d. Moreover we are able to prove a bound on the number of decision procedure queries.
- Using the observation that a boolean combination of linear equalities with d disjunctions (in DNF form) is equivalent to an algebraic invariant of degree d [17,26], we describe an algorithm to generate an equivalent linear invariant from an algebraic invariant.
- We evaluate our technique on benchmark programs from various papers on generation of algebraic loop invariants and our results are encouraging—starting with a small amount of data, Guess-and-Check terminates on all benchmarks in one iteration, that is, our first guess is an actual invariant.

The remainder of the paper is organized as follows. Section 2 motivates and informally illustrates the Guess-and-Check algorithm over an example program. Section 3 introduces the background for the technical material in the paper. Section 4 presents the Guess-and-Check algorithm for algebraic invariants and also proves its correctness and termination. Section 5 describes some extensions: our technique for obtaining disjunctive linear invariants from algebraic invariants and a discussion about richer theories such as arrays. Section 6 evaluates our implementation of the GUESS-AND-CHECK algorithm on several benchmarks for algebraic loop invariants. Section 7 surveys related work and, finally, Section 8 concludes the paper.

2 Overview of the Technique

We will illustrate our technique over the example program shown in Figure 1. Our objective is to compute the loop invariant for the loop in this program. Informally, a loop invariant over-approximates the set of all possible program states that are possible at a loop head. This method can be generalized to obtain invariants at any program point. This program has a loop (lines 3 – 6) that is non-deterministic. In line 2 and 6, we have instrumentation code that writes the program state (the values of the variables x and y) to a log file. The loop invariant for this program is $I \equiv y + y^2 = 2x$. Since our approach is data driven, the starting point is to run the program with test inputs and accumulate the resulting data (in other words, the resulting program states) in a log. Assume that the program execution exercises the loop once. On such an execution, we obtain program states $x = y = 0$ and $x = y = 1$.

```
1:  assume(x=0 && y=0);
2:  writelog(x, y);
3:  while (nondet()) do
4:    y := y+1;
5:    x := x+y;
6:    writelog(x, y);
```

Fig. 1. Example for algebraic invariants.

```
1:  if (x >= 0) then y := x
2:  else y := -x;
3:  writelog(x, y);
4:  while (y>=0 && nondet()) do
5:    if(x >= y) then
6:              y := y+1; x := x+1;
7:    else y := y+1; x := x-1;
8:    writelog(x, y);
```

Fig. 2. Example for (disjunctive) linear invariants.

It turns out that for our technique to work, we need to assume an upper bound d on the degree of the polynomials that constitute the invariant. For this example, we assume that $d = 2$, which allows us to exhaustively enumerate all the monomials over the program variables up to the chosen degree. For our example, $\vec{\alpha} = \{1, x, y, y^2, x^2, xy\}$ is the set of all monomials over the variables x and y with degree less than or equal to 2. The number of monomials of degree d in n variables is large: $\binom{n+d-1}{d}$. Heuristics exist to discard the monomials that are unlikely to be a part of an invariant [24].

Using $\vec{\alpha}$ and the program states, we construct a *data matrix* A that is a 2×6 matrix with one row corresponding to each program state and six columns, one for each monomial in $\vec{\alpha}$. Every entry in j^{th} column of A represents the value of the j^{th} monomial over the program execution. Therefore,

$$A = \begin{array}{|c|c|c|c|c|c|}\hline 1 & x & y & y^2 & x^2 & xy \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array} \quad (1)$$

As we will see in Section 4.1, we can employ the null space of A to compute a candidate invariant I as follows. If $\{b_1, b_2, \ldots, b_k\}$ is a basis for the null space of the data matrix A, then

$$I \equiv \bigwedge_{i=1}^{k} ([1, x, y, y^2, x^2, xy] b_i = 0) \quad (2)$$

is a candidate invariant that is logically stronger than the strongest algebraic invariant. The null space of A is defined by four basis vectors, representing four algebraic equations:

$$I \equiv x = y \wedge x = y^2 \wedge x = x^2 \wedge x = xy \quad (3)$$

Next, in the check phase, we check whether I as specified by Equation 3 is actually an invariant. Abstractly, if $L \equiv$ `while` B `do` S is a loop, then to check if I is a loop invariant, we need to establish the following conditions:

1. If φ is a precondition at the beginning of L, then $\varphi \Rightarrow I$.
2. Furthermore, executing the loop body S with a state satisfying $I \wedge B$, always results in a state satisfying the invariant I.

The above checks for validating I are performed by an off-the-shelf decision procedure [16]. For our example, we first check whether the precondition at the beginning of the loop implies I:

$$(x = 0 \wedge y = 0) \Rightarrow (x = y = x^2 = y^2 = xy)$$

This condition is indeed valid, and therefore we check whether I is inductive (we obtain the predicate representing the loop body via symbolic execution [14]):

$$((x = y = x^2 = y^2 = xy) \wedge y' = y + 1 \wedge x' = x + y') \Rightarrow (x' = y' = x'^2 = y'^2 = x'y')$$

This predicate is not valid, and we obtain a counterexample $x' = 3$, $y' = 2$ at line 3 of the program. Let us assume that we generate more program states by executing the loop for three iterations and starting with $x = 3$ and $y = 2$. As a result, we get a data matrix (that also includes the rows from the previous data matrix) as shown:

$$A = \begin{array}{|c|c|c|c|c|c|}\hline 1 & x & y & y^2 & x^2 & xy \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 4 & 9 & 8 \\ 1 & 6 & 3 & 9 & 36 & 18 \\ 1 & 10 & 4 & 16 & 100 & 40 \\ \hline \end{array} \quad (4)$$

As with the earlier iteration, we require the basis of the null space of A and this is defined by the single vector: $[0, 2, -1, -1, 0, 0]$. Therefore, from Equation 2, it follows that the candidate invariant is $I \equiv 2x - y - y^2 = 0$.

Now, the conditions that must hold for I to be a loop invariant are:

1. $(x = 0 \wedge y = 0) \Rightarrow y + y^2 = 2x$, and
2. $(y + y^2 = 2x \wedge y' = y + 1 \wedge x' = x + y') \Rightarrow (y' + y'^2 = 2x')$

both of which are deemed to be valid by the check phase, and therefore $I \equiv y + y^2 = 2x$ is the desired loop invariant.

Following a similar approach, we can infer the algebraic invariant $x^2 = y^2$ for Figure 2. In Section 5.1, we show a data-driven procedure to generate equivalent linear invariants from algebraic invariants and use the same to infer the linear invariant $y = x \vee y = -x$ for Figure 2.

3 Preliminaries

We consider programs belonging to the following language of *while programs*:

$$\mathcal{S} \;::=\; x{:=}M \mid \mathcal{S};\, \mathcal{S} \mid \texttt{if } B \texttt{ then } \mathcal{S} \texttt{ else } \mathcal{S} \mid \texttt{while } B \texttt{ do } \mathcal{S}$$

where x is a variable over a countably infinite sort *loc* of memory locations, M is an expression, and B is a boolean expression. Expressions in this language are either of type `int` or `bool`.

A *monomial* α over the variables $\vec{x} = x_1, \dots x_n$ is a term of the form $\alpha(\vec{x}) = x_1^{k_1} x_2^{k_2} \dots x_n^{k_n}$. The *degree* of a monomial is $\sum_{i=1}^{n} k_i$. A *polynomial* $f(x_1, \dots, x_n)$ defined over n variables $\vec{x} = x_1, \dots, x_n$ is a weighted sum of monomials and has the following form.

$$f(\vec{x}) = \sum_k w_k x_1^{k_1} x_2^{k_2} \dots x_n^{k_n} = \sum_k w_k \alpha_k \tag{5}$$

where $\alpha_k = x_1^{k_1} x_2^{k_2} \dots x_n^{k_n}$ is a monomial. We are interested in polynomials over rationals, that is, $\forall k \;.\; w_k \in \mathbb{Q}$. The *degree* of a polynomial is the maximum degree over its constituent monomials: $max_k \; \{degree(\alpha_k) \mid w_k \neq 0\}$.

An *algebraic equation* is of the form $f(\vec{x}) = 0$, where f is a polynomial. Given a loop $L = \texttt{while } B \texttt{ do } S$ defined over variables $\vec{x} = x_1, \dots, x_n$ together with a precondition φ, a *loop invariant* I is the strongest predicate such that $\varphi \Rightarrow I$ and $\{I \wedge B\}S\{I\}$. Any predicate I satisfying these two conditions is an invariant for L. If we do not impose the condition that we need the strongest invariant, then the trivial predicate $I = true$ is a valid invariant. In this section, we will focus on *algebraic invariants* for a loop. An algebraic invariant $\mathcal{I}$ is of the form $\wedge_i f_i(\vec{x}) = 0$, where each f_i is a polynomial over the variables $\vec{x}$ of the loop.

3.1 Matrix Algebra

This section reviews basic linear algebra. Readers familiar with matrix algebra may safely skip this section.

The *span* of a set of vectors $\{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathbb{Q}^m$, is the set of all vectors that can be expressed as a linear combination of $\{x_1, x_2, \ldots, x_n\}$. Therefore,

$$span(\{x_1, x_2, \ldots, x_n\}) = \{v \mid v = \sum_{i=1}^{n} \alpha_i x_i, \alpha_i \in \mathbb{Q}\} \tag{6}$$

For any $P = span(x_1, \ldots, x_n) \subseteq \mathbb{Q}^m$, if every vector $v \in P$ can be written as a linear combination of vectors from a linearly independent set $B = \{b_1, b_2, \ldots, b_k\}$, and B is minimal, then B forms a *basis* of P, and k is called the *dimension* of the set P.

The *range* of a matrix $A \in \mathbb{Q}^{m \times n}$ is the span of the columns of A. That is,

$$range(A) = \{v \in \mathbb{Q}^m \mid v = Ax, x \in \mathbb{Q}^n\} \tag{7}$$

The dimension of $range(A)$ is called $rank(A)$. The *null space* of a matrix $A \in \mathbb{Q}^{m \times n}$ is the set of all vectors that equal to 0 when multiplied by A. More precisely,

$$NullSpace(A) = \{x \in \mathbb{Q}^n \mid Ax = 0\} \tag{8}$$

The dimension of $NullSpace(A)$ is called its *nullity*. For instance, the matrix

$$A = \begin{bmatrix} 1 & 2 & -3 \\ 3 & 5 & 9 \\ 5 & 9 & 3 \end{bmatrix} \text{ has a null space spanned by } \left\{ \begin{bmatrix} -33 \\ 18 \\ 1 \end{bmatrix} \right\} \text{ with } nullity(A) = 1.$$

A basis for the null space of a $m \times n$ matrix can be computed in time $O(m^2 n)$. A *subspace* of $\mathbb{Q}^n$, spanned by a basis B, are the vectors x that satisfy $Ax = 0$, where A is a basis for the null space of B. From the fundamental theorem of linear algebra, for any matrix $A \in \mathbb{Q}^{m \times n}$, we know

$$rank(A) + nullity(A) = n \tag{9}$$

4 The Guess-and-Check Algorithm

The Guess-and-Check algorithm is described in Figure 3. The algorithm takes as input a while program L, a precondition φ on the inputs to L, and an upper bound d on the degree of the desired invariant, and returns an algebraic loop invariant $\mathcal{I}$. If $L =$ `while` B `do` S, then recall that $\mathcal{I}$ is the strongest predicate such that

$$\varphi \Rightarrow \mathcal{I} \text{ and } \{\mathcal{I} \wedge B\}S\{\mathcal{I}\} \tag{10}$$

As the name suggests, Guess-and-Check consists of two phases.

1. *Guess phase* : this phase processes the data in the form of concrete program states at the loop head to compute a data matrix, and uses linear algebra techniques to compute a candidate invariant.

```
GUESS-AND-CHECK($L$,$\varphi$,$d$)
Returns: A loop invariant $\mathcal{I}$ for $L$.
 1: $\vec{x} := vars(L)$
 2: $Tests := TestGen(\varphi, L)$
 3: $logfile := \langle\rangle$
 4: for $\vec{t} \in Tests$ do
 5:     $logfile := logfile :: Execute(L, \vec{x} = \vec{t})$
 6: end for
 7: repeat
 8:     $\mathcal{I} := Guess(logfile, d)$
 9:     $(done, \vec{t}) := Check(\mathcal{I}, L, \varphi)$
10:     if $\neg done$ then
11:         $logfile := logfile :: t$
12:     end if
13: until $done$
14: return $\mathcal{I}$
```

```
Guess($logfile$,$d$)
Returns: A candidate invariant
 1: if $logfile = \langle\rangle$ then
 2:     return $false$
 3: end if
 4: $A := DataMatrix(logfile, d)$
 5: $\mathcal{B} := Basis(NullSpace(A))$
 6: if $\mathcal{B} = \emptyset$ then
 7:     // No non-trivial invariant
 8:     return $true$
 9: end if
10: return $CandidateInvariant(\mathcal{B})$
```

Fig. 3. GUESS-AND-CHECK computes an algebraic invariant of degree d for an input while program L with a precondition φ

2. *Check phase* (line 9): this phase uses an off-the-shelf decision procedure for checking if the candidate invariant computed in the guess phase is indeed a true invariant (using the conditions in Equation 10) [16].

The GUESS-AND-CHECK algorithm works as follows. In line 1, $\vec{x}$ represents the input variables of the while program L. The procedure *TestGen* is any test generation technique that generates a set of test inputs *Tests* that satisfy the precondition φ. Alternatively, our technique could also employ an existing test suite for *Tests*. The variable *logfile* maintains a sequence of concrete program states at the loop head of L. Line 3 initializes *logfile* to the empty sequence. Lines 4–13 perform the main computation of the algorithm. First, the program L is executed over every test $\vec{t} \in Tests$ via the call to *Execute* in line 5. *Execute* runs a loop till termination (or for a timeout to avoid non-terminating executions) on a test input and generates a sequence of states at the loop head. E.g., $Execute(\texttt{while}(x! = 0)\ \texttt{do}\ x--, x = 2)$ will generate states $\{x = 2, x = 1, x = 0\}$ for the data matrix. Note that this sequence also include the states that violate the loop guard. The call to *Guess* (line 8) constructs a matrix A with one row for every program state in *logfile* and one column for every monomial from the set of all monomials over $\vec{x}$ whose degree is bounded above by d (as informally illustrated in Section 2). The $(i, j)^{th}$ entry of A is the value of the j^{th} monomial evaluated over the program state represented by the i^{th} row.

Next, using off-the-shelf linear algebra solvers, we compute the basis for the null space of A. If $\mathcal{B}$ is empty, then this means that there is no algebraic equation, of given degree d, that the data satisfies and we return *true*. Otherwise, the candidate invariant represented by $\mathcal{B}$ is given to the checking procedure *Check* in line 9. The procedure *Check* uses off-the-shelf SMT solvers [16] to check whether the candidate invariant $\mathcal{I}$ satisfies the conditions in Equation 10. If so, then $\mathcal{I}$ is an invariant and the procedure terminates by returning $\mathcal{I}$. Otherwise, *Check* returns a counter-example in the form of a test input $\vec{t}$ that explains why $\mathcal{I}$ is not an invariant – the computation is repeated with this new test input $\vec{t}$, and the process continues until we have found an invariant. Note that, as in Section 2, we can also add the states generated by $Execute(L, \vec{x} = \vec{t})$ to *logfile* (instead of just adding $\vec{t}$). In either case, the size of *logfile* strictly increases in every iteration.

In summary, the guess and check phases of Guess-and-Check operate iteratively, and in each iteration if the actual invariant cannot be derived, then the algorithm automatically figures out the reason for this and corrective measures are taken in the form of generating more test inputs (this corresponds to the case where the data generated is insufficient for guessing a sound invariant). In the next section, we will formally show the correctness of the Guess-and-Check algorithm – we prove that it is a sound and we bound the number of iterations of Guess-and-Check (the loop consisting of lines 7 to 13 of Figure 3).

4.1 Connections between Null Spaces and Invariants

In the previous section, we have seen how Guess-and-Check computes an algebraic invariant over monomials $\vec{\alpha}$ that consist of all monomials over the variables of the input while program with degree bounded above by d. The starting point for proving correctness of Guess-and-Check is the data matrix A as computed in line 1 of *Guess* procedure of Figure 3.

An invariant $\mathcal{I} \equiv \wedge_i^k (w_i^T \vec{\alpha} = 0)$ has the property that for each w_i, $1 \leq i \leq k$, $w_i^T a_j = 0$ for each row $a_j \in \mathbb{Q}^n$ of A – in other words, $Aw_i = 0$. This shows that each w_i is a vector in the null space of the data matrix A. Conversely, any vector in $NullSpace(A)$ is a reasonable candidate for being a part of an invariant.

We make the observation that a candidate invariant will be a true invariant if the dimension of the space spanned by the set $\{w_i\}_{1 \leq i \leq k}$ equals $nullity(A)$. We will assume, without loss of generality, that $\{w_i\}_{1 \leq i \leq k}$ is a linearly independent set. Then, by definition, the dimension of the space spanned by $\{w_i\}_{1 \leq i \leq k}$ is k.

Consider an n-dimensional space where each axis corresponds to a monomial of $\vec{\alpha}$. Then the rows of the matrix A are points in this n-dimensional space. Now assume that $w^T \vec{\alpha} = 0$ is an invariant, that is, $k = 1$. This means that all rows a_j of A satisfy $w^T a_j = 0$. In particular, the points corresponding to the rows of A lie on an $n-1$ dimensional subspace defined by $w^T \vec{\alpha} = 0$. If the data or program states generated by the test inputs *Tests* (line 2 in Figure 3) is insufficient, then A might not have rows spanning the $n-1$ dimensions. Therefore, from Equation 9, we have $n - rank(A) = nullity(A) \geq 1$ if the invariant is a single algebraic equation. Generalizing this, we can say that $nullity(A)$ is an upper bound on the number of algebraic equations in the invariant. The following lemma and theorem formalize this intuition.

Lemma 1 (Invariant is in null space). *If $\wedge_i^k w_i^T \vec{\alpha} = 0$ is an invariant, and A is the data matrix, then all w_i lie in $NullSpace(A)$.*

Proof. This follows from the fact that for every w_i, $1 \leq i \leq k$, $Aw_i = 0$.

Therefore, the null space of the data matrix A gives us the subspace in which the invariants lie. In particular, if we arrange the vectors that form the basis for *NullSpace*(A) as columns in a matrix V, then *range*(V) defines the space of candidate invariants.

Theorem 1. *If* $\wedge_{i=1}^{k} w_i^T \vec{\alpha} = 0$ *is an invariant with the set* $\{w_1, w_2, \ldots, w_k\}$ *forming a linearly independent set,* A *is the data matrix and nullity*$(A) = k$*, then any basis for NullSpace*(A) *forms an invariant.*

Proof. Let $B = [v_1 \cdots v_k]$ be a matrix with each v_i, $1 \leq i \leq k$ being a column vector, and with *span*$(\{v_1, \ldots, v_k\})$ equal to *NullSpace*(A). That is, $\{v_1, \ldots, v_k\}$ is a basis for *NullSpace*(A). From Lemma 1, we know that every w_i, $1 \leq i \leq k$, lies in *span*$(\{v_1, \ldots, v_k\})$. This means that every w_i, $1 \leq i \leq k$, can be written as $w_i = Bu_i$ for some vector $u_i \in \mathbb{Q}^k$. Therefore, if $B^T \vec{\alpha} = 0$ then $u_i^T B^T \vec{\alpha} = 0$, which implies that $w_i^T \vec{\alpha} = 0$, $1 \leq i \leq k$.

Observe that $\{w_1, w_2, \ldots, w_k\}$ form a basis for *NullSpace*(A), and therefore every v_j, $1 \leq j \leq k$, can be written as a linear combination of vectors from $\{w_1, w_2, \ldots, w_k\}$. From this, it follows that $\wedge_{i=1}^{k} w_i^T \vec{\alpha} = 0 \implies v_j^T \vec{\alpha} = 0$ for all $1 \leq j \leq k$. Thus, $\wedge_{i=1}^{k} w_i^T \vec{\alpha} = 0 \Leftrightarrow \wedge_{j=1}^{k} v_j^T \vec{\alpha} = 0$.

Theorem 1 precisely defines the implementation of the "guess" step. Furthermore, Theorem 1 also states that we need to have enough data represented by the data matrix A so that *nullity*(A) equals k, the dimension of the space spanned by $\{w_i\}_{1 \leq i \leq k}$. If this is indeed the case, then $\mathcal{I} \equiv \wedge_{j=1}^{k} v_j^T \vec{\alpha} = 0$ will be an invariant. On the other hand, if the data is not enough, then Lemma 1 guarantees that the candidate invariant $\mathcal{I}$ is a sound under-approximation of the loop invariant. If the null space is zero-dimensional, then only the trivial invariant *true* constitutes an invariant over conjunction of polynomial equations that has degree less than or equal to d.

The question of how much data must be generated in order to attain *nullity*(A) $= k$ is an empirical one. In our experiments, we were able to generate invariants using a relatively small data matrix for various benchmarks from the literature.

4.2 Check Candidate Invariants

Computing the null space of the data matrix provides us a way for proposing candidate invariants. The candidates are complete; they do not miss any algebraic equations. But they might be unsound. They might contain spurious equations. To obtain soundness, we will use a decision procedure analogous to the technique proposed in [25].

Theorem 2 (Soundness). *If the algorithm* GUESS-AND-CHECK *terminates and the underlying decision procedure for checking candidate invariants (Check) is sound, then it returns an invariant.*

Next, we prove that the algorithm GUESS-AND-CHECK terminates.

Theorem 3 (Termination). *If the underlying decision procedure Check is sound and complete, then the algorithm* GUESS-AND-CHECK *will terminate after at most n iterations, where n is the total number of monomials whose degree is bounded by d.*

Proof. Let $A \in \mathbb{Q}^{m \times n}$ be the data matrix computed in line 4 in the *Guess* procedure of Figure 3. If the candidate invariant $\mathcal{I}$ computed in line 8 of GUESS-AND-CHECK is an invariant (that is, *done* = *true*), then GUESS-AND-CHECK terminates.

Therefore, let us assume that $\mathcal{I}$ is not an invariant, and let $\vec{t}$ be the test or counterexample that violates the candidate invariant as computed in line 9 of the algorithm. As a result, GUESS-AND-CHECK adds $\vec{t}$ to A – call the resulting matrix $\hat{A}$. By construction, we also know that $\vec{t} \notin range(A^T)$. Therefore, it follows that $rank(\hat{A}) = rank(A) + 1$. More generally, adding a counter-example to the data matrix A necessarily increases its rank by 1. From Equation 9, we know that the rank of A is bounded above by n, which implies that GUESS-AND-CHECK will terminate in at most n iterations.

Note that since we are concerned with integer manipulating programs, a sound and complete decision procedure for *Check* cannot exist: the queries are in Peano arithmetic which is undecidable. However, for our experiments, we found that the `Z3` [16] SMT solver sufficed (see Section 6). `Z3` has limited support for non-linear integer arithmetic: It combines extensions on top of simplex and reduction to SAT (after bounding) for these queries. One might try to achieve completeness for GUESS-AND-CHECK by giving up soundness. Just as [23,4,21], if we interpret program variables as real numbers then `Z3` does have a sound and complete decision procedure for non-linear real arithmetic [12] that has been demonstrated to be practical. Since `Z3` supports both non-linear integer and real arithmetic, we can easily combine or switch between the two, if desired (see Section 6).

4.3 Nested Loops

GUESS-AND-CHECK easily extends to nested loops, while maintaining soundness and termination properties. Given a program with M loops, we construct data matrices for each loop. Let the number of columns of the data matrix of i^{th} loop be denoted by n_i. We run tests and generate candidate invariants $\vec{I}$ at all loop heads. Next, the candidate invariants are checked simultaneously. For checking the candidate invariant of an outer loop, the inner loop is replaced by its candidate invariant and a constraint is generated. For checking the inner loop, the candidate invariant of the outer loop is used to compute a pre-condition. If a counter-example is obtained then it generates more data and invariant computation is repeated. We continue these guess and check iterations until the check phase passes for all the loops; thus, on termination the output consists of sound invariants for all loops. Also, the initial candidate invariants $\vec{I}$ are under-approximations of the actual invariants by Lemma 1, a property that is maintained throughout the procedure and allows us to conclude that when the procedure terminates the output invariants are the strongest possible over algebraic equations. To prove termination, note that each failed decision procedure query increases the rank of some data matrix for some loop, which implies that the number of decision procedure queries which can fail is bounded by $\sum_{i=1}^{M} n_i$. Hence, if $N = max\ n_i$ then the total number of decision procedure queries is bounded by M^2N.

5 Extensions

In this section we discuss two extensions of our technique. We first discuss how algebraic invariants can be converted to equivalent linear invariants. Then we discuss how our approach can be extended to compute invariants over more expressive theories, such as the theory of arrays.

5.1 From Algebraic to Linear Invariants

Conventional invariant generation techniques for linear equalities [13] do not handle disjunctions. Using disjunctive completion to obtain disjunctions of equalities entails a careful design of the widening operator. Techniques for generation of non-linear invariants can generate algebraic invariants that are equivalent to a boolean combination of linear equalities. But if these invariants are to be consumed by a tool that understands only linear arithmetic, it is important to obtain the original linear invariant from the algebraic invariant. For example, verification engines like [10] are based on linear arithmetic and cannot use non-linear predicates for predicate abstraction. It is not obvious how this step can be performed since the discovered polynomials might not factor into linear factors.

Since our approach is data driven, we can solve this problem using standard machine learning techniques. Here is another perspective on converting algebraic to linear invariants. Assume that the algebraic invariant is equivalent to a boolean combination of linear equalities. Express this linear invariant in DNF form. For instance, for the program in Figure 2, we have the DNF formula $y = -x \vee y = x$. The rows of the data matrix A are satisfying assignments of this DNF formula. Hence, each row satisfies some disjunct: each row of A satisfies $y = -x$ or $y = x$. If we create partitions of our data such that the states in each partition satisfy the same disjunct, then all the states of a single partition will lie on a subspace: they will satisfy some conjunction of linear equalities. The aim is to find the subspaces in which the states lie. Since a subspace represents a conjunction of linear equalities, a disjunction of all such subspaces can represent an invariant that is a boolean combination of linear equalities.

The problem of obtaining boolean combinations of linear equalities that a given data matrix satisfies is called subspace segmentation in the machine learning community. This problem arises in applications such as face clustering, video segmentation, motion segmentation, and several algorithms have been proposed over the years. In this section we will apply the algorithm of Vidal, Ma, and

Sastry [26] to obtain linear invariants from algebraic invariants. The main insight is that the derivative of the polynomials constituting the algebraic invariant evaluated at a program state characterizes the subspace in which the state lies.

The derivative of the polynomial corresponding to the algebraic invariant for Figure 2, that is, $x^2 - y^2$ is $[2x, -2y]$: the first entry is partial derivative w.r.t. x and the second entry is the partial derivative w.r.t. y. Running the program with test input $x \in \{-1, 1\}$ for say 4 iterations each will results in a data matrix A with 10 rows. The first and last rows are shown:

$$A = \begin{array}{|c|c|c|c|c|c|}\hline 1 & x & y & y^2 & x^2 & xy \\ \hline 1 & -1 & 1 & 1 & 1 & -1 \\ 1 & 5 & 5 & 25 & 25 & 25 \\ \hline \end{array} \tag{11}$$

Evaluating the derivative at first state of A gives us $[-2, -2]$. This shows that the first state belongs to $-2x - 2y = 0$ i.e. $x = -y$. Evaluating at the last state gives us $[10, -10]$, which shows that the last state belongs to $10x - 10y = 0$ or $x = y$. The other 8 states of A (not shown in Equation 11) also belong to $x = y$ or $x = -y$ and we return the disjunction of these two predicates as the candidate invariant. The relationship between the boolean structure of a linear invariant and its equivalent algebraic invariant can be described as follows: the number of conjunctions in the linear invariant (in CNF form) corresponds to the number of conjunctions in the algebraic invariant, and the number of disjunctions in the linear invariant (in DNF form) corresponds to the degree of the algebraic invariant.

Now we explain why this approach works. We sketch the proof from [26] for the case when there is a single algebraic equation $f(\vec{x}) = 0$, that is, the invariant is a disjunction of linear equalities. The case of multiple algebraic equations is similar. Say the invariant is $\vee_i w_i^T \vec{x} = 0 \Leftrightarrow \left(\prod_i w_i^T \vec{x}\right) = 0 \equiv f(\vec{x}) = 0$. The derivative of $f(\vec{x})$, denoted by $\nabla f(\vec{x})$, is a vector of $|\vec{x}|$ elements where the l^{th} element of the vector is a partial derivative with respect to the l^{th} variable:

$$(\nabla f(\vec{x}))_l = \frac{\partial f(\vec{x})}{\partial x_l}.$$

Now using,

$$\nabla \left(f(\vec{x})g(\vec{x})\right) = \left(\nabla f(\vec{x})\right) g(\vec{x}) + f(\vec{x}) \left(\nabla g(\vec{x})\right)$$

and

$$\nabla w^T \vec{x} = \left[\frac{\partial w_1 x_1}{\partial x_1}, \ldots, \frac{\partial w_n x_n}{\partial x_n}\right]^T = w \text{ where } |\vec{x}| = n$$

we obtain:

$$\nabla f(\vec{x}) = \nabla \left(\prod_i w_i^T \vec{x}\right) = \sum_i w_i \prod_{j \neq i} (w_j^T \vec{x})$$

Say a program state a satisfies $w_k^T a = 0$. Then $(\nabla f)(a)$ is a scalar multiple of w_k because $\prod_{j \neq i} w_j^T a = 0$ for $i \neq k$. Hence evaluating the derivative at a program state provides the subspace in which the state lies. For more details see [26].

Next we remove from A the states that lie in the same subspace. Next, if A still contains a program state then we can repeat by finding the derivative at that state. In the end we get a collection of subspaces that contain every state of the original data-matrix. A union of these subspaces gives us a boolean combination of linear equalities.

Theorem 4. *Given an algebraic invariant $\mathcal{I}$ = NullSpace(A) equivalent to a linear invariant, the procedure of [26] finds a linear invariant equivalent to $\mathcal{I}$.*

Note that this conversion is unsound if no equivalent linear invariant exists. Hence the linear predicate should be checked for equivalence with the algebraic invariant; this check can be performed using a decision procedure. Note that we are able to easily incorporate the technique of [26] with Guess-and-Check as our technique is data driven. Also, this conversion just requires differentiating polynomials symbolically, that can be performed linearly in the size of the invariant, and evaluating the derivative at all points in the data matrix. The latter operation is just a matrix multiplication. Hence this algorithm is quite efficient.

5.2 Richer Theories

An interesting question is whether the algorithm Guess-and-Check generalizes to richer theories beyond polynomial arithmetic. It is indeed possible and requires careful design of the representation of data. For instance, if we want to infer invariants in the theory of linear arithmetic and arrays, we can have an additional column in the data matrix for values obtained from arrays. Similarly, we can have a variable that stores the value returned from an uninterpreted function and assign it a column in the data matrix. Hence it is possible to use our technique to infer conjunctions of equalities in richer theories too if we know the constituents of the invariants, analogous to invariant generation techniques based on templates.

In order to illustrate how the Guess-and-Check technique would work for programs with arrays, consider the example program shown in Figure 4. We want to prove that the assertion in line 6 holds for all inputs to the program. Assume that we log the values of $a[i]$ and i after every iteration and that the degree bound is $d = 1$. The data matrix that Guess-and-Check constructs has three columns and let us assume that we run a single test with input $n = 1$ resulting in rows corresponding to program states induced by this input at the loop head. The data matrix A is shown in Figure 5. The null space of A is defined by the basis vector $B = [0, 1, -1]^T$, and therefore we obtain the invariant $a[i] = i$ that is sufficient to prove that the assertion holds.

Our approach of using a dynamic analysis technique to generate data in the form of concrete program states and augmenting it with a decision procedure to obtain a sound technique is a general one. Similar ideas have been used for computing interpolants [25]. We can also take the method for discovering array invariants or polynomial inequalities of [18] and extend it to a sound procedure in a similar fashion.

```
(i,a[0]) = (0,0);
assume (n > 0);
while (i != n) do
  i := i+1;
  a[i] := a[i-1]+1;
done
assert(a[n] == n);
```

Fig. 4. Example with arrays

$$A = \begin{array}{|c|c|c|}\hline 1 & i & a[i] \\ \hline 1 & 0 & 0 \\ 1 & 1 & 1 \\ \hline \end{array}$$

Fig. 5. Data for `n`=1

Table 1. `Name` is the name of the benchmark; `#vars` is the number of variables in the benchmark; `deg` is the user specified maximum possible degree of the discovered invariant; `Data` is the number of times the loop under consideration is executed over all tests; `#and` is the number of algebraic equalities in the discovered invariant; `Guess` is the time taken by the guess phase of GUESS-AND-CHECK in seconds. `Check` is the time in seconds taken by the check phase of GUESS-AND-CHECK to verify that the candidate invariant is actually an invariant. The last column represents the total time.

`Name`	`#vars`	`deg`	`Data`	`#and`	`Guess` (s)	`Check` (s)	`Total` (s)
`Mul2` [23]	4	2	75	1	0.0007	0.010	0.0107
`LCM/GCD` [23]	6	2	329	1	0.004	0.012	0.016
`Div` [23]	6	2	343	3	0.454	0.134	0.588
`Bezout` [21]	8	2	362	5	0.765	0.149	0.914
`Factor` [21]	5	3	100	1	0.002	0.010	0.012
`Prod` [22]	5	2	84	1	0.0007	0.011	0.0117
`Petter` [22]	2	6	10	1	0.0003	0.012	0.0123
`Dijkstra` [22]	6	2	362	1	0.003	0.015	0.018
`Cubes` [20].	4	3	31	10	0.014	0.062	0.076
`geoReihe1` [20]	3	2	25	1	0.0003	0.010	0.0103
`geoReihe2` [20]	3	2	25	1	0.0004	0.017	0.0174
`geoReihe3` [20]	4	3	125	1	0.001	0.010	0.011
`potSumm1` [20]	2	1	10	1	0.0002	0.011	0.0112
`potSumm2` [20]	2	2	10	1	0.0002	0.009	0.0092
`potSumm3` [20]	2	3	10	1	0.0002	0.012	0.0122
`potSumm4` [20]	2	4	10	1	0.0002	0.010	0.0102

6 Experimental Evaluation

We evaluate the GUESS-AND-CHECK algorithm on a number of benchmarks from the literature. All experiments were performed on a 2.5GHz Intel i5 processor system with 4 GB RAM running Ubuntu 10.04 LTS.

Benchmarks. The benchmarks over which we evaluated the GUESS-AND-CHECK algorithm are from a number of recent research papers on inferring algebraic invariants [21,22,23]. These are shown in the first column of Table 1. These programs were implemented in C for data generation.

Evaluation. We now describe our implementation and our experimental results of Table 1. For a detailed description of the implementation please see [24]. The second column of Table 1 shows the number of variables in each benchmark program. The third column shows the given upper bound for the degree of the polynomials in the inferred invariant.

The fourth column shows the number of rows of the data matrix. The data or tests are generated naively; each input variable is allowed to take values from 1 to N where N is between 5 and 20 for the experiments. Hence if there are two input variables we have N^2 tests. These tests are executed till termination to generate data. While it is possible to generate tests more intelligently, using inputs from a very small bounding box demonstrates the generality of our technique by not tying it to any symbolic execution engine. Note that including all the states reaching the loop head, over all tests, can include redundant states that do not affect the output. Since the algorithms for null space computation are quite efficient, we do not attempt to identify and remove redundant states. If needed, heuristics like considering a random subset of the states [18] can be employed to keep the size of data matrices small. The fifth column shows the number of algebraic equations in the discovered loop invariant. For most of the programs, a single algebraic equation was sufficient. The null space and the basis computations were performed using off-the-shelf linear algebra algorithms in MATLAB. Guess-and-Check finds invariants equivalent to those reported in the earlier papers [20,21,22,23]. The time (in seconds) taken by the guess phase of Guess-and-Check is reported in the sixth column of Table 1.

We use `Z3` [16] for checking that the proposed invariants are actually invariants (implementation of *Check* procedure in the Guess-and-Check algorithm). Theorem prover `Z3` was able to easily handle the simple queries made by Guess-and-Check, because once an invariant has been obtained, the constraint encoding that the invariant is inductive is quite a simple constraint to solve and our naively generated tests were sufficient to generate an actual invariant. For all programs, except `Div`, we declare the variables as integers. So even though these queries are in Peano arithmetic, and can contain integer division and modulo operators, the decision procedure is able to discharge them. For `Div` the invariant that [23] finds is inductive only if the variables are over reals. When we execute Guess-and-Check on `Div`, where the queries are in Peano arithmetic, we obtain the trivial invariant *true* after three guess-and-check iterations. Next, we lift the variables to reals when querying `Z3`. Now, we discover the invariant found by [23] in one guess-and-check iteration and this is the result shown in Table 1. By the soundness of our approach, we conclude that an approach producing a non-trivial algebraic invariant for this benchmark can be unsound for integer manipulating programs containing division or modulo operators.

Finally, the time taken by Guess-and-Check on these benchmarks is comparable to the state-of-the-art correct-by-construction invariant generation techniques [4]. Since these benchmarks are small and the time taken by both our technique and [4] is less than a second on these programs, a comparison of run times may not be indicative of performance of either approach on larger loops.

For these benchmarks, our algorithm is significantly faster than any algorithm using Gröbner bases. For instance, on the benchmark `factor`, [22] takes 55.4 seconds, while [21] takes 2.61 seconds. We discover the same invariant in 0.012 seconds. However, the exact timings must be taken with a grain of salt (we are running on a newer generation of hardware). See Section 7 for a more detailed comparison with the previous work. We leave the collection of a hard benchmark suite for algebraic invariant generation tools as future work.

7 Related Work

We now place the Guess-and-Check algorithm in the context of existing work on discovering algebraic loop invariants. Major benefits of our data driven approach include finding sound invariants for integer manipulating programs, consuming a rich syntax (depends only on the decision procedure), and extracting linear invariants from algebraic invariants in time linear in the data, that increases the applicability of our algorithm. Sankaranarayanan et al. [23] describe a constraint based technique that uses user-defined templates for computing algebraic invariants. Their objective is to find an instantiation of these templates that satisfies the constraints and results in an invariant. The constraints they use contain quantifiers and therefore the cost of solving them is quite high.

Abstract interpretation based techniques either ignore [15,22] or restrict conditions on branches to equalities or dis-equalities [17,6,21,4]. The techniques of [6,21,22,15] use Gröbner bases computations and [17] has no upper complexity bound. Cachera et al. [4] provide an algorithm that does not use Gröbner bases but interprets variables as taking values over the real numbers. In contrast, we handle programs with division and modulo operations soundly. Bagnara et al. [3] introduce new variables for monomials and generate linear invariants over them by abstract interpretation. Amato et al. [2] analyze data from program executions to tune their abstract interpretation.

Nguyen et al. [18] have proposed a dynamic analysis for inference of candidate invariants. They do not provide any formal characterization of the output of their algorithm and do not prove any soundness and completeness theorems. The Daikon tool [9] generates likely invariants from tests and templates. Our approach is similar in that it is also based on analyzing data from tests. Daikon does not provide any formal guarantees such as soundness and completeness over the invariants it generates. In the context of Daikon, it is interesting to note from [19] that very few test cases suffice for invariant generation. Indeed, this has been our experience with Guess-and-Check as well.

8 Conclusion

We have presented a sound data driven algorithm for discovering algebraic equation invariants. We use linear algebra techniques to guess an invariant from the data generated from program runs, and use decision procedures for non-linear arithmetic to validate these candidate invariants.

We are able to formally prove that the guessed invariant under-approximates the actual invariant, as well as bound the number of iterations of Guess-and-Check. Thus, the key novelty of the Guess-and-Check approach is the data driven analysis together with formal guarantees of soundness and termination. Our guarantees are stronger than some of the previous techniques, since we do not lift integral variables of programs to reals. Moreover, the data driven approach facilitates transformation of algebraic invariants to linear invariants. We have also informally shown how our approach can be extended to more expressive theories such as arrays.

We have implemented the Guess-and-Check algorithm and evaluated it on a number of benchmarks from recent papers on invariant generation and our results are encouraging. Future work includes incorporating the Guess-and-Check algorithm into a mainstream program verification engine [10] that can consume the candidate invariants as relevant predicates for proofs and a bug-finding engine [1] that can use the candidate invariants to abstract loops by their sound under-approximations and obtain better coverage. Since these tools generally work over linear arithmetic, the transformation from algebraic to linear invariants will play a critical role.

Acknowledgements. We thank the anonymous reviewers for their constructive comments. We thank Divya Gupta and Hristo Paskov for helpful discussions.

References

1. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: PASTE, pp. 43–48 (2007)
2. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. J. Symb. Comput. 47(12), 1533–1560 (2012)
3. Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 19–34. Springer, Heidelberg (2005)
4. Cachera, D., Jensen, T., Jobin, A., Kirchner, F.: Inference of Polynomial Invariants for Imperative Programs: A Farewell to Gröbner Bases. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 58–74. Springer, Heidelberg (2012)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
6. Colón, M.A.: Approximating the Algebraic Relational Semantics of Imperative Programs. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 296–311. Springer, Heidelberg (2004)
7. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)

10. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE, pp. 117–127 (2006)
11. Gupta, A., Majumdar, R., Rybalchenko, A.: From Tests to Proofs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 262–276. Springer, Heidelberg (2009)
12. Jovanović, D., de Moura, L.: Solving Non-linear Arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 339–354. Springer, Heidelberg (2012)
13. Karr, M.: Affine relationships among variables of a program. Acta Inf. 6, 133–151 (1976)
14. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
15. Kovács, L.: A Complete Invariant Generation Approach for P-solvable Loops. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 242–256. Springer, Heidelberg (2010)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Inf. Process. Lett. 91(5), 233–244 (2004)
18. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE, pp. 683–693 (2012)
19. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: ISSTA, pp. 229–239 (2002)
20. Petter, M.: Berechnung von polynomiellen invarianten. Master's thesis, Fakultät für Informatik, Technische Universität München (2004)
21. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Science of Computer Programming 64(1), 54–75 (2007)
22. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443–476 (2007)
23. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL, pp. 318–329 (2004)
24. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: A data driven approach for algebraic loop invariants. Tech. Report MSR-TR-2012-97, Microsoft Research (2012)
25. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as Classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
26. Vidal, R., Ma, Y., Sastry, S.: Generalized principal component analysis (GPCA). IEEE Trans. Pattern Anal. Mach. Intell. 27(12), 1945–1959 (2005)