Termination Proofs from Tests

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Rahul Sharma[*]
Stanford University
sharmar@stanford.com

ABSTRACT

We show how a test suite for a sequential program can be profitably used to construct a termination proof. In particular, we describe an algorithm TPT for proving termination of a program based on information derived from testing it.

TPT iteratively calls two phases: (a) an infer phase, and (b) a validate phase. In the infer phase, machine learning, in particular, linear regression is used to efficiently compute a candidate loop bound for every loop in the program. These loop bounds are verified for correctness by an off-the-shelf checker. If a loop bound is invalid, then the safety checker provides a test or a counterexample that is used to generate more data which is subsequently used by the next infer phase to compute better estimates for loop bounds. On the other hand, if all loop bounds are valid, then we have a proof of termination. We also describe a simple extension to our approach that allows us to infer polynomial loop bounds automatically.

We have evaluated TPT on two benchmark sets, micro-benchmarks obtained from recent literature on program termination, and Windows device drivers. Our results are promising – on the micro-benchmarks, we show that TPT is able to prove termination on 15% more benchmarks than any previously known technique, and our evaluation on Windows device drivers demonstrates TPT's ability to analyze and scale to real world applications.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Statistical methods

General Terms

Testing, Verification

[*]This author performed the work reported here during a summer internship at Microsoft Research India.

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia
Copyright 13 ACM 978-1-4503-2237-9/13/08 ...$15.00.

Keywords

Machine learning; Software model checking; Termination

1. INTRODUCTION

Proving termination of sequential programs is a hard and important problem whose solutions can broadly improve software reliability and programmer productivity. There is a huge body of work that is based on a number of interesting techniques such as abstract interpretation [4, 7, 39], bounds analysis [17, 18], ranking functions [5, 6, 10], recurrence sets [20, 22] and transition invariants [25, 32]. In spite of these techniques, much progress has to be made to design an efficient and scalable technique for program termination.

In the world of safety checking, there are a number of efficient and scalable techniques that use tests for proving safety properties of programs [3,13,16,21,28,35–37]. This is advantageous as most programs have test suites that ensure quality, and more importantly, these tests are a rich source of reachability information (in other words, information from tests gives reachability information for free without the need for any static analysis).

Unfortunately, there are no analogous techniques for proving termination or liveness properties of programs. We ask the following question: *Can a test suite for a program be used profitably for constructing termination proofs?* In this paper, we answer this question in the affirmative, and present an algorithm TPT that uses information from tests to construct a sound proof of termination of a program.

TPT is based on the insight that information from tests can be used to efficiently learn loop bounds for every loop in a program W. If every loop in W has a sound loop bound, then this is a proof of termination of W. The algorithm iteratively calls two phases: (a) an infer phase, and (b) a validate phase. In the infer phase, machine learning, in particular, linear regression is used to efficiently compute a candidate loop bound for every loop in the program. These loop bounds are verified for correctness by an off-the-shelf checker. If a loop bound is invalid, then the safety checker provides a test or a counterexample that is used to generate more data which is subsequently used by the next infer phase. On the other hand, if all loop bounds are valid, then we have a proof of termination. This separation of the infer phase from the validate phase allows our approach to handle more expressive syntax than previous approaches for termination of numerical programs. Being driven by tests also allows us to easily extend TPT to infer polynomial loop bounds automatically (see Section 5). TPT is easy to implement. In fact, the infer phase of TPT is implemented by

```
1 gcd(int x, int y)
2 {
3   assume(x>0 && y>0);
4   while (x != y) {
5     if (x > y) x = x-y;
6     if (y > x) y = y-x;
7   }
8   return x;
9 }
```

Figure 1: Computing GCD.

```
 1 gcd(int x, int y)
 2 {
 3   assume(x>0 && y>0);
 4   // instrumented code
 5   a = x; b = y; c = 0;
 6   while (x != y) {
 7     // instrumented code
 8     c = c+1;
 9     writeLog(a, b, c, x, y);
10     if (x > y) x = x-y;
11     if (y > x) y = y-x;
12   }
13   return x;
14 }
```

Figure 2: Instrumented GCD program.

```
 1 gcd(int x, int y)
 2 {
 3   assume(x>0 && y>0);
 4   a = x; b = y; c = 0;
 5   while (x != y) {
 6     // annotation
 7     free_invariant(c <= a+b-x-y);
 8     // annotation
 9     assert(c <= a+b-2);
10     c = c+1;
11     if (x > y) x = x-y;
12     if (y > x) y = y-x;
13   }
14   return x;
15 }
```

Figure 3: Annotated GCD program.

just one line of MATLAB code, while the validate phase is an off-the-shelf safety checker.

The contributions of this paper are as follows:

- We describe TpT, a novel termination prover for sequential programs that is based on information derived from tests. This information is analyzed to automatically learn loop bounds.

- We present an empirical evaluation of TpT on various benchmarks from recent papers on program termination and our results are encouraging. On these benchmarks, TpT is able to prove termination on 15% more benchmarks than any previously known technique. We also evaluate TpT on Windows device drivers and the results are positive. There are two drivers in this set of drivers for which TpT is able to successfully prove termination, whereas the Terminator tool [9] fails. This demonstrates TpT's ability to analyze and scale to real world applications.

The rest of the paper is organized as follows: Section 2 informally illustrates the TpT algorithm with the help of a motivating example. In Section 3 we review background material necessary for this paper. Section 4 formally describes the TpT algorithm, and Section 5 extends the algorithm to the non-linear case. Section 6 describes our experimental evaluation. In Section 7, we discuss related work, and finally, Section 8 concludes the paper.

2. OVERVIEW OF THE TECHNIQUE

Let $L =$ `while` B `do` S be a loop defined over the variables $x_1, \ldots, x_n$. Our goal is to find an expression $\tau(x_1, \ldots, x_n)$ defined over the initial values of the variables $x_1, \ldots, x_n$ such that the maximum number of iterations of L over all possible values of $x_1, \ldots, x_n$ is bounded above by $\tau(x_1, \ldots, x_n)$. It is easy to see that such a bound $\tau(x_1, \ldots, x_n)$ will imply termination of L. Therefore, for an arbitrary program, a termination proof is a loop bound τ_i for every loop L_i in the program.

We will illustrate our technique on the `gcd` program shown in Figure 1. We also assume the availability of a test suite Γ for this program. This program has a loop (lines 4 – 7) which performs the greatest common divisor (GCD) computation. Our objective is to automatically compute an upper bound on the number of iterations of the loop (over all inputs x, y to the program) by testing the program. To collect data from these tests, we instrument the program – the instrumented GCD program is shown in Figure 2. The ghost variables variables a and b (in line 5) record the initial values of the variables x and y respectively and are not modified by the loop. The variable c is a loop counter which is initialized to zero in line 5.

The loop counter c is incremented in every iteration of the loop (line 8). The values of the variables a, b, c, x and y are recorded at the beginning of the loop in line 9 via the call to the function `writeLog`. The function `writeLog` records the program state (specifically, the values of variables that are its arguments) to a log file. We want to bound the number of loop iterations c by a linear expression $\tau(a, b)$ over the inputs a and b. In particular, we want to compute $w_1, w_2, w_3 \in \mathbb{R}$ such that $\tau(a, b) = w_1 a + w_2 b + w_3$, and $c \leq \tau(a, b)$.

Next, TpT executes the instrumented program over test inputs, and accumulates the resulting data via calls to the logging function `writeLog`. Using the log file, TpT constructs two *data matrices* A and C defined as follows: In addition to the all-ones column (the first column), the matrix A contains the test input values to the program in each row (one row for every program state generated before executing the loop body) and the matrix C contains the corresponding value of the loop counter c. Assume that we test the program on inputs $(x, y) = \{(1, 2), (2, 1), (1, 3), (3, 1)\}$. As a result, we have:

$$A = \begin{array}{|c|c|c|} \hline 1 & a & b \\ \hline 1 & 1 & 2 \\ \hline 1 & 2 & 1 \\ \hline 1 & 1 & 3 \\ 1 & 1 & 3 \\ \hline 1 & 3 & 1 \\ 1 & 3 & 1 \\ \hline \end{array} \quad C = \begin{array}{|c|} \hline c \\ \hline 1 \\ \hline 1 \\ \hline 1 \\ 2 \\ \hline 1 \\ 2 \\ \hline \end{array} \tag{1}$$

The rows generated by each input have been partitioned. Each input generates as many rows as the number of iterations. The problem of learning an upper bound on the loop counter c can be looked upon as a *linear regression* problem in machine learning [27]. In linear regression, we are given a set of input-output pairs as input, and the goal is to learn a function which maps the inputs to the outputs. In our setting, the inputs are the rows of A, the outputs are the corresponding rows of C, and we want to learn a function B that maps the rows of A to the rows of C.

For our purpose, we use a variant of linear regression to learn a bound on c. Conventional linear regression aims to find $w = (w_1, w_2, w_3)$ such that $c \approx w_1 a + w_2 b + w_3$ (c is a linear function of the parameters w which is why the regression is called "linear"). This can be obtained by solving the following optimization problem:

$$min \quad \textstyle\sum_i (w_1 a_i + w_2 b_i + w_3 - c_i)^2 \tag{2}$$

where $(1, a_i, b_i)$ is the i^{th} row of A, and c_i is the i^{th} row of C. Since we want an upper bound on c, we add additional constraints $w_1 a_i + w_2 b_i + w_3 \geq c_i$, for all i – this results in the following quadratic programming problem [29]:

$$\begin{array}{rl} min & \sum_i (w_1 a_i + w_2 b_i + w_3 - c_i)^2 \\ s.t. & Aw \geq C \end{array}$$

For the matrices A and C in Equation 1, $(w_1, w_2, w_3) = (1, 1, -2)$ is a solution to this optimization problem. Therefore TpT returns $\tau(a, b) = a + b - 2$ as the candidate loop bound.

Next, in the validate phase, we check whether $\tau(a, b)$ is indeed a valid loop bound. Since the tests represent an under-approximation of program behaviors, it is possible that there is a yet unseen test that executes the loop for more than $\tau(a, b)$ iterations. To handle this, we annotate the program with an assertion encoding the soundness of this bound and check it using an off-the-shelf safety checker. For our example, the annotated program is shown in Figure 3. If the safety checker is able to verify the assertion of line 9, then we have verified that the program terminates.

For proving soundness of the loop bound, a safety checker will also need a loop invariant. Unfortunately, these invariants can be quite hard to infer. For our example, the loop invariant is a bound on the loop counter c that is a linear expression over the values of the program variables a, b, x and y. Once again, this is a linear regression problem. Using the log file generated by the instrumented program in Figure 2 on inputs $(x, y) = \{(1, 2), (2, 1), (1, 3), (3, 1)\}$, we construct the following matrices:

$$\hat{A} = \begin{array}{|c|c|c|c|c|} \hline 1 & \mathrm{a} & \mathrm{b} & \mathrm{x} & \mathrm{y} \\ \hline 1 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 3 \\ 1 & 1 & 3 & 1 & 2 \\ 1 & 3 & 1 & 3 & 1 \\ 1 & 3 & 1 & 2 & 1 \\ \hline \end{array} \quad \hat{C} = \begin{array}{|c|} \hline c \\ \hline 1 \\ 1 \\ 1 \\ 2 \\ 1 \\ 2 \\ \hline \end{array} \tag{3}$$

Matrix $\hat{A}$ is the same as A in Equation 1, except that it also contains values for the program variables x and y. The problem of computing a loop invariant translates to the following linear regression problem: find $w = (w_1, w_2, w_3, w_4, w_5)$ such that $\iota(a, b, x, y) = w_1 a + w_2 b + w_3 x + w_4 y + w_5$, and $c \leq \iota(a, b, x, y)$. Thus we want to solve the following optimization problem:

$$\begin{array}{rl} min & \sum_i (w_1 a_i + w_2 b_i + w_3 x_i + w_4 y_i - c_i)^2 \\ s.t. & \hat{A}w \geq \hat{C} \end{array}$$

This optimization problem is also a quadratic program which can be efficiently solved using an off-the-shelf solver. For our example, we obtain $\iota(a, b, x, y) = a + b - x - y$, and thus $c \leq a + b - x - y$ is the desired loop invariant. Once again, since this invariant has been computed from an under-approximation of program behaviors, it is just a candidate invariant and thus has to be checked for validity. Therefore, this inferred invariant is marked as a "free invariant" [2] in line 7 of Figure 3. A free invariant is just a candidate invariant that can potentially be useful to the safety checker. If the free invariant is invalid, then the safety checker can simply ignore it. On the other hand, if the safety checker is able to prove that the free invariant is an invariant, then it can use this candidate invariant to prove the assertion.

Indeed, in our experiments (see Section 6), on all the benchmarks for which TpT is able to prove termination, the free invariants were actual invariants. However, it is often the case that these candidate invariants had to be strengthened by our safety checker to make them inductive. For the program in Figure 3, we observe that our safety checker uses the free invariant to obtain the following inductive invariant:

$$c \leq a + b - x - y \wedge x > 0 \wedge y > 0 \tag{4}$$

This invariant is sufficient to prove the assertion of Figure 3, and therefore, TpT is able to prove the termination of the `gcd` program in Figure 1.

3. PRELIMINARIES

W.l.o.g., we consider *while programs* W which are defined by the following grammar:

$$\begin{array}{rcl} W & ::= & x{:=}E_a \\ & & \mid W; W \\ & & \mid \texttt{if } E_b \texttt{ then } W \texttt{ else } W \texttt{ fi} \\ & & \mid \texttt{while } E_b \texttt{ do } W \end{array}$$

where x is a variable, E_a is an arithmetic expression, and E_b is a boolean expression.

3.1 Algebra

A *monomial* $\alpha : x_1 \times \ldots \times x_n \to \mathbb{R}$ is an expression of the form $\alpha(x_1, \ldots, x_n) = x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n}$. The *degree* of a monomial $x_1^{k_1} x_2^{k_2} \dots x_n^{k_n}$ is defined as $\sum_{i=1}^{n} k_i$. A *polynomial* $f : x_1 \times \ldots \times x_n \to \mathbb{R}$ is a weighted sum of monomials:

$$f(x_1, \ldots, x_n) = \sum_k w_k x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} = \sum_k w_k \alpha_k \qquad (5)$$

where $\alpha_k = x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n}$ is a monomial. The *degree* of a polynomial is the maximum degree over its constituent monomials:

$$degree(f) \overset{\mathsf{def}}{=} max_k \; \{degree(\alpha_k) \mid w_k \neq 0\} \qquad (6)$$

Let $A \in \mathbb{R}^{m \times n}$ be a *matrix* defined over the set of real numbers $\mathbb{R}$ with m rows and n columns. Let $x \in \mathbb{R}^n$ denote a *vector*. The vector x can also be represented as a row matrix $x^T \in \mathbb{R}^{1 \times n}$, where T is the matrix transpose operator. In general, the transpose of a matrix A, denoted A^T is obtained by interchanging the rows and columns of A. In other words, $\forall A \in \mathbb{R}^{m \times n}$, $(A^T)_{ij} = A_{ji}$.

Given two matrices $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$, their product AB is a matrix $C \in \mathbb{R}^{m \times n}$ with each entry C_{ij} defined as follows:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \qquad (7)$$

The *inner product* $\langle x, y \rangle$ of two vectors x and y is the product of the matrices corresponding to x^T and y which is defined as:

$$x^T y = \sum_{i=1}^{n} x_i y_i \qquad (8)$$

3.2 Linear Regression and Quadratic Programming

Given a matrix $X \in \mathbb{R}^{m \times n}$ and a column vector $Y \in \mathbb{R}^m$, *linear regression* is the problem of learning a function f such that $f(X_i) = Y_i$, where X_i is the i^{th} row of X and Y_i is the i^{th} row of Y. Specifically, the function f is a "linear" expression over model parameters $\theta \in \mathbb{R}^n$ such that for each i, $f(X_i) = \langle \theta, X_i \rangle$, and $Y_i = f(X_i)$. In general, such a function f might not exist, in which case linear regression aims to compute a function f such that $f(X_i)$ best approximates Y_i. This approximation is formally characterized by the following optimization problem:

$$min \; \sum_i (X_i^T \theta - Y_i)^2 \qquad (9)$$

In other words, we would like to minimize a quadratic function of θ. From an implementation point of view (see Section 6), it is convenient to write Equation 9 in matrix notation as follows:

$$min \; \frac{1}{2} \theta^T X^T X \theta - \theta^T (X^T Y) \qquad (10)$$

Consider the data points (represented by +) shown in Figure 4. These data points correspond to rows of the matrix X in Equation 10. Linear regression attempts to find a "best-fit" line (as defined by the optimization problem in Equation 10) for these points – specifically, linear regression finds θ_0 and θ_1 such that $y = \theta_0 + \theta_1 x$ is the best linear approximation of f. As we will see in Section 4.1, for proving termination, it is useful to further constrain the parameters θ by inequality constraints $X\theta \geq Y$ as follows:

$$\begin{array}{ll} min & \frac{1}{2}\theta^T X^T X \theta - \theta^T (X^T Y) \\ s.t. & X\theta \geq Y \end{array} \qquad (11)$$

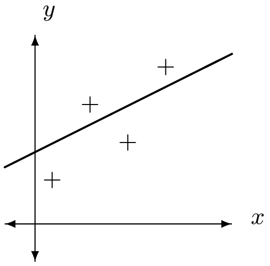

Figure 4: Finding the best-fit line using linear regression.

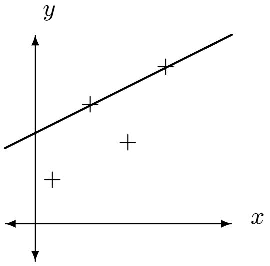

Figure 5: A solution to the quadratic program in Equation 11.

Equation 11 is an instance of quadratic programming [29] which is defined as follows:

$$\begin{array}{ll} min & \frac{1}{2}\theta^T H \theta + \theta^T f \\ s.t. & M\theta \leq N \end{array} \qquad (12)$$

By replacing H wth $X^T X$, f with $-X^T Y$, M with $-X$, and N with $-Y$ in Equation 12, we obtain the constrained linear regression problem in Equation 11. For the data in Figure 4, solving the quadratic program in Equation 11 results in the line shown in Figure 5. An interesting feature of this solution line is that $f(x) \geq y$ for all data points (x, y), which is precisely the property that we require for estimating loop bounds.

4. ALGORITHM

The TPT algorithm is described in Figure 6. The algorithm takes a while program W as input together with a test suite Γ for W. Assume that the program W has l (possibly nested) loops $L_1, \ldots, L_l$. For every loop L_k, $1 \leq k \leq l$, TPT returns a loop bound τ_k. Note that a valid loop bound for each loop in W defines a proof of termination for W. As previously illustrated by Figure 2, in line 1, the call to the function $Instrument_{data}$ instruments the program so that information from testing the program W is recorded to a log file. The program W is instrumented as follows:

Consider a loop L from $\{L_k\}_{k=1}^{l}$. Let $x_1, \ldots, x_n$ be the variables in the scope of the loop L. For each x_i, $1 \leq i \leq n$, create a ghost variable g_i, and let c be a variable representing a loop counter. Every ghost variable g_i records the value of its corresponding variable x_i before the program enters the loop L. The loop counter c is also initialized to zero before the program enters L. At the beginning of each iteration of the loop L, the values of all the variables in the scope of L (including the newly introduced ghost variables and loop counter) are written to a log. The value of c is also incremented before writing to the log ensuring that it is incremented in every iteration of the loop. Formally, if $L \equiv$

```
TpT($W$: While-Program, $\Gamma$: Test-Suite)
Returns: A loop bound $\tau$ for $W$.

 1: $W_I := Instrument_{data}(W)$
 2: $logfile := Test(W_I, \Gamma)$
 3: $\Delta := \emptyset$
 4: repeat
 5:    $logfile := logfile :: Test(W_I, \Delta)$
 6:    $\{(A_k, C_k)\}_{k=1}^l := DataMatrix_{LoopBound}(logfile)$
 7:    $\{(\hat{A}_k, \hat{C}_k)\}_{k=1}^l := DataMatrix_{FreeInvariant}(logfile)$
 8:    $\{\tau_k\}_{k=1}^l := QP(\{(A_k, C_k)\}_{i=k}^l)$
 9:    $\{\iota_k\}_{k=1}^l := QP(\{\hat{A}_k, \hat{C}_k\}_{k=1}^l)$
10:    for $\tau \in \{\tau_k\}_{k=1}^l$ do
11:       if $Fail(\tau)$ then
12:          $\tau := RoundOrPartition(\tau, \{(A_k, C_k)\}_{k=1}^l)$
13:       end if
14:    end for
15:    $W_{check} := Instrument_{check}(W, \{\tau_k\}_{k=1}^l, \{\iota_k\}_{k=1}^l)$
16:    $(done, \Delta) := Check(W_{check})$
17: until $done$
18: return $\{\tau_i\}_{i=1}^l$
```

Figure 6: The TpT algorithm for a while program.

`while` E_b `do` S `od`, then the instrumented loop L_I is defined as follows:

```
$g_i := x_i,\ 1 \le i \le n$;
$c := 0$;
while $E_b$ do
   $c := c + 1$;                                      (13)
   writeLog($g_1, \ldots, g_n, x_1, \ldots, x_n, c$);
   $S$
od
```

Therefore, the instrumented program W_I is the program W with all loops instrumented as described in Equation 13. The call to `writeLog` returns a sequence of concrete states. In this case, a concrete state maps all the variables passed as arguments to `writeLog` to their corresponding values.

The program W_I is first executed over all tests in the test suite Γ (line 2) and the results are stored in *logfile*. If *Test* executes W_I, on some test, for more iterations than a user specified time-out then the loop is classified as "potentially non-terminating" and the proof fails. Lines 4–18 perform the main computation of the algorithm which primarily consists of two phases (analogous to the abstraction and refinement phases of CEGAR based model checking tools [8]):

1. *Infer phase* (lines 5–14): this phase processes *logfile* to compute the data matrices A_i and C_i for every loop L_i, using which a candidate loop bound τ_i is estimated (line 6), as well as the data matrices $\hat{A}_i$ and $\hat{C}_i$ using which the free invariant ι_i is estimated (line 7).

2. *Validate phase* (line 15–16): this phase uses an off-the-shelf safety checker for checking the validity of the candidate loop bounds computed in the infer phase.

The function $DataMatrix_{LoopBound}$ (line 6) constructs from *logfile* two matrices A_k and C_k for every loop L_k in the program W. Let $\sigma(L_k)$ denote the total number of times the entry of the loop L_k is executed over all tests in the test suite Γ. Every row in the matrix A_k represents a concrete state for all the ghost variables in L_k. Therefore, the $(i, j)^{th}$ entry of A_k is the value of the j^{th} ghost variable obtained when the entry of L_k is executed the i^{th} time by tests from Γ. The i^{th} entry of the matrix C_k is the corresponding value of the loop counter for the same execution. Therefore, A_k has dimension $\sigma(L_k)$ by the number of ghost variables for L_k, and C_k has dimension $\sigma(L_k)$ by one.

The function $DataMatrix_{FreeInvariant}$ (line 7) constructs from *logfile* two matrices $\hat{A}_k$ and $\hat{C}_k$ for every loop L_k in the program W. Every row in the matrix $\hat{A}_k$ represents a concrete state for all variables (including the ghost variables) in L_k. Therefore, the $(i, j)^{th}$ entry of $\hat{A}_k$ is the value of the j^{th} variable obtained when the entry of L_k is executed the i^{th} time over all tests from Γ. The i^{th} entry of the matrix $\hat{C}_k$ is the corresponding value of the loop counter for the same test. Therefore, $\hat{A}_k$ has dimension $\sigma(L_k)$ by the number of variables for L_k, and $\hat{C}_k$ has dimension $\sigma(L_k)$ by one. To summarize, A_k is a sub-matrix of $\hat{A}_k$ and $\hat{C}_k \equiv C_k$.

Next, in line 8, the call to the function *QP* solves l quadratic programming instances, one for each loop L_k, $1 \le k \le l$ in the program W. This is done by using an off-the-shelf quadratic programming solver for each instance, thereby computing a candidate loop bound τ_k for each loop L_k. As we will see in Section 4.1, the candidate loop bound τ_k can be efficiently computed from A_k and C_k. If the loop bound computation fails (lines 11 – 13), then appropriate corrective measures are taken (details in Section 4.3). The loop bound computation is said to have *failed* when the bound obtained does not have integral coefficients. The computation in line 9 is similar to that in line 8, but results in the calculation of the free invariant for every loop. The candidate loop bounds $\{\tau_k\}_{k=1}^l$ together with the free invariants $\{\iota_k\}_{k=1}^l$ are given to the checking procedure *Check* in line 16. The procedure *Check* uses an off-the-shelf safety checker to check whether the instrumented program W_{check} is correct. If $L \equiv$ `while` E_b `do` S `od` is a loop in W with candidate loop bound τ, free invariant ι, and variables in scope $x_1, \ldots, x_n$, then the instrumented loop L_{check} is defined as follows:

```
$g_i := x_i,\ 1 \le i \le n$;
$c := 0$;
while $B$ do
   free_invariant($c \le \iota$)
   assert($c \le \tau$);                              (14)
   $c := c + 1$;
   $S$
od
```

The result W_{check} of the function call $Instrument_{check}$ in line 15 is the program W with all loops instrumented as described in Equation 14.

If the program W_{check} is safe, then τ_k for every loop L_k is an upper bound, and TpT terminates by returning a termination proof $\{\tau_k\}_{k=1}^l$. Otherwise, *Check* returns a set Δ of counterexamples or tests that explain why some candidate loops bounds are not valid – the computation in lines 5 – 16 is repeated with these new tests Δ (line 5), and the process continues until we have found a sound upper bound for every loop in W.

To summarize, the infer and validate phases of TpT operate iteratively, and in each iteration if the actual bound for every loop cannot be derived, then the algorithm automatically figures out the reason for this, and appropriate action in the form of corrective measures (Section 4.3) or generating more tests (this corresponds to the case where the data generated is insufficient) is taken.

4.1 The Infer Phase

Let L be a loop, and let A and C be its corresponding data matrices for the loop bound calculation (computed in line 6 of Figure 6). Let the data matrix $A = [a_1, \ldots, a_m]^T$, and the data matrix $C = [c_1, \ldots, c_m]^T$, then any valid loop bound $\tau \equiv \langle w, g \rangle$ over ghost variables $g = [g_1, \ldots g_n]$, for L, with $w \in \mathbb{R}^n$, is such that:

$$Aw \geq C \tag{15}$$

We would also like our candidate loop bound τ for L to be predictive. In other words, if we run the program on more test inputs, then the bound should still be satisfied. Therefore, we employ machine learning techniques for generating a predictive bound. We describe these techniques next.

Linear regression finds a w such that $Aw \approx C$. It has been successfully used to derive predictive answers in a variety of applications. More formally, linear regression finds a w which minimizes the following quantity:

$$min \sum_{1 \leq i \leq m} (a_i^T w - c_i)^2 \tag{16}$$

In our setting, using a w computed by linear regression naively has the following problem: There is no guarantee that w is an upper bound. That is, it is possible that for some rows a_i^T of A, $a_i^T w < c_i$. This motivates the need for a constrained linear regression, that is, minimize the expression in Equation 16 subject to the constraints $Aw \geq C$. In other words, we are interested in solving the following quadratic program:

$$\begin{array}{ll} min & \frac{1}{2} w^T A^T A w - w^T A^T C \\ s.t. & Aw \geq C \end{array} \tag{17}$$

Upon solving the quadratic program, we are able to compute an upper bound that is consistent with the current set of concrete program states. The techniques described above are also useful for efficiently generating a free invariant for a loop.

4.2 The Validate Phase

After a candidate bound for every loop in the program has been obtained, TpT asks an off-the-shelf safety checker to validate these bounds. To reduce the burden of invariant inference on the safety checker, TpT also provides it with free invariants which are computed by the infer phase. It is important to note that the free invariants only serve as hints to the safety checker. If a free invariant is invalid, then the safety checker ignores it. In Section 6, we show empirical data that establishes the usefulness of free invariants obtained via linear regression of test data. If a loop bound is invalid, then the safety checker returns a counterexample or a test that demonstrates the violation of the assertion. This set of tests (denoted by Δ in Figure 6) is used to generate more concrete states and this ensures better loop bounds in the next iteration of TpT.

4.3 Corrective Measures

There can be several reasons for the failure of bound generation (line 11 in Figure 6). The first reason is that the bound obtained for a loop cannot be expressed as a linear expression with integer coefficients. We describe a simple rounding strategy to handle this issue. If a variable has value between $N - 0.1$ and $N + 0.1$ (where N is an integer), then we round it to N. If a variable still does not have an integral coefficient, then we check if it is constrained by the program to be positive or negative. If yes, then we round it "soundly": ceiling for positive input variables and floor for negative input variables. This is sound because increasing the coefficient of positive variables can only make the bound looser, and similarly for the negative variables. Hence the rounded bounds can be imprecise, but they are consistent with the data and are sufficient to prove termination. In our experiments, out of all the benchmarks TpT succeeds on, rounding was performed only in four cases. For the rest, the output obtained by solving the quadratic program of Equation 17 was already integral.

If rounding fails, then the reason might be that the loop does not have a single bound. The loop might show different behaviors for different inputs, and hence a single bound is perhaps not possible for this mix of two or more behaviors. Therefore, there is a need to partition the input values to the loop such that for each region in the partition, the tests in that region have the same bound. Once a correct partitioning has been obtained, linear regression can be used to obtain the bounds for each individual partition. We consider two simple schemes for partitioning the data:

1. Partition according to the predicates syntactically occurring in the program. For instance, if the guard for a loop is $p \vee q$, then we can partition the input values based on whether they satisfy the predicate p, q, or both. [10,11].

2. Partition according to the path executed in running the test. There are existing techniques that are based on control flow refinement [17,34] to do this systematically. In [34], if there is an "if" condition in the loop that shows "phases" (it has the property that once the condition becomes *false*, it remains *false* until the program exits the loop) then the loop is rewritten into a sequence of two loops: the first containing the "then" branch and the second containing the "else" branch. Using information from tests, it is easy to detect if a condition is showing phases.

If the above heuristics fail, that is, either there is no obvious partitioning by predicates, or branches showing phases, then TpT fails. If even after partitioning, we obtain coefficients for which the above rounding process fails, then we declare failure of TpT. These heuristics for partitioning are basically trying to handle disjunctive bounds. How to split the proof burdens for termination is a well known open problem [4]. Our technique is good at obtaining linear or nonlinear bounds when they exist. Handling disjunctive bounds systematically is left for future work. The failure to infer the correct partition is the primary reason why this technique can fail. In our benchmarks, for two benchmarks we performed predicate based splitting, and for one benchmark we needed path based splitting. We illustrate partitioning by predicates with the help of the example program shown in Figure 7. We can partition the tests into three regions, each defined by the predicates $a < M \wedge b < N$, $a < M \wedge b \geq N$, $a \geq M \wedge b < N$. By applying the infer phase to the tests

```
a = i; b = j; c = 0;
while (i < M || j < N) {
   i = i+1;
   j = j+1;
   c = c+1;
}
```

Figure 7: Program that requires partitioning.

```
u = x; v = y; w = z; c = 0;
while (x >= y) {
   c = c+1;
   if (z > 0) {
      z = z-1;
      x = x+z;
   } else
      y = y+1;
}
```

Figure 8: Example program that requires a non-linear loop bound.

based on each of these regions separately, we obtain:

$$\begin{array}{l} a < M \wedge b < N \Rightarrow c \leq M + N - a - b \\ a < M \wedge b \geq N \Rightarrow c \leq M - a \\ a \geq M \wedge b < N \Rightarrow c \leq N - b \end{array} \tag{18}$$

The safety checker is able to prove the disjunctive loop bound in Equation 18. Using clustering algorithms [27] in machine learning for inferring disjunctive loop bounds systematically is left for future work.

5. NON-LINEAR BOUNDS

Since the TPT algorithm is driven by tests, it is able to seamlessly handle non-linearities. We illustrate this using the example program in Figure 8.

Let d be the degree of the polynomial τ representing the loop bound. Assume that d is bounded above by 2 for this example. As before, just as in the linear case, we create the data matrices A and C:

- The matrix A is such that the i^{th} row of A corresponds to the i^{th} concrete program state. Each column of A corresponds to a monomial of degree at most two over the variables u, v and w (where u, v and w are ghost variables recording the initial values of the program variables x, y and z respectively). Therefore, A has 10 columns, one for each of the monomials 1, u, v, w, uu, vv, ww, uv, uw and vw. For instance, when the program is tested on the input $x = y = z = 1$, it generates a row of A with ten ones.

- The i^{th} row of the matrix C is the number of iterations for the i^{th} execution of L_k.

By way of random testing, where the inputs x, y and z to the program are in the range $[-4, 4]$, and only test inputs that execute the loop at least once are selected, we obtain 288 tests (essentially, this is a form of rejection sampling [33] to select tests). Thus A is a 288×10 matrix, and C is a 288×1 matrix. Solving the quadratic program as defined by Equation 17, we obtain the solution:

$$w^T = [1.9, 1, -1, 0.95, 0, 0, 0.24, 0, 0, 0] \tag{19}$$

This gives us a loop bound $\tau(u, w) = 2 + u - v + w + w^2 \implies c \leq 2 + u - v + w + w^2$ after applying rounding (see Section 4.3 for a description of rounding).

Next, using the data from the logfile, we compute the free invariant $c \leq y + w - v - z$ (see Section 4). This free invariant together with the assertion encoding the candidate loop bound is fed to a safety checker, which is able to prove the bound and therefore termination of this program.

6. EXPERIMENTAL EVALUATION

We have evaluated the TPT algorithm on various benchmarks for termination from [4, 7] as well as some Windows device drivers. All experiments were performed on a 2.67GHz Intel Xeon processor system with 8 GB RAM running Windows 7 and MATLAB R2012a. The infer phase of TPT, in particular, Equation 11 is implemented using just one line of MATLAB code:

$$\texttt{quadprog}(\texttt{A}^{\texttt{T}} * \texttt{A}, -\texttt{A}^{\texttt{T}} * \texttt{C}, -\texttt{A}, -\texttt{C})$$

where A, C are the data matrices for a loop.

Micro-Benchmarks.

These benchmarks are shown in the first column of Table 1, and are categorized as follows:

1. Benchmarks prefixed by `Oct` are from the OCTANAL distribution. These include programs such as heapsort and bubblesort.

2. Benchmarks prefixed by `Driver` are code fragments from Windows device drivers. These benchmarks have been hand-translated to remove pointer aliasing.

3. Benchmarks prefixed by `Poly` are from the POLYRANK distribution. These benchmarks are tricky programs, such as McCarthy's 91 function, with non-trivial termination arguments.

Device Drivers.

We also evaluated TPT on Windows Device Drivers to demonstrate its ability to analyze and scale to real world applications. These drivers are part of the SDV-RP toolkit (available at `http://research.microsoft.com/slam`). The statistics for these drivers are shown in columns 2 and 3 of Table 3. The column `LOC` is the number of lines of code, and the column `#Loops` is the number of loops that are reachable from the harness to these drivers. We performed a simple static analysis to determine this loop reachability information.

Evaluation.

The second column of Table 1 shows the total time spent by TPT in its infer phase. The third column shows the total time taken by the validate phase of TPT. The fourth column shows the total time (in seconds) taken by TPT. We observe that a significant fraction of the total analysis time is spent in the validate phase. For the micro-benchmarks, the data or tests were generated naively; each input variable was allowed to take values from $-N$ to N, where N was between 5 and 20. Therefore, if a program has two input variables, we generated $O(N^2)$ tests. If the benchmark program had nondeterministic branches, then we ran

Name	Infer time (sec)	Validate time (sec)	Total time (sec)	Result
Oct1	0.004	0.98	0.98	✓
Oct2	0.007	1.00	1.01	✓
Oct3	0.081	0.98	1.06	✓
Oct4	0.004	0.95	0.95	✓
Oct5	0.002	0.92	0.92	✓
Oct6	0.002	0.95	0.95	✓
Driver1	0.007	1.91	1.92	✓
Driver2	0.003	2.67	2.67	×
Driver3	0.001	2.26	2.26	×
Driver4	0.001	2.20	2.20	✓
Driver5	0.0003	0.94	0.94	✓
Driver6	0.004	1.02	1.02	✓
Driver7	0.006	1.01	1.02	✓
Driver8	0.006	1.00	1.01	✓
Driver9	0.001	0.94	0.94	×
Driver10	0.18	14.69	14.87	✓
Poly1	0.24	10.11	10.35	✓
Poly2	0.017	0.95	0.97	✓
Poly3	0.035	1.33	1.37	✓
Poly4	FAIL	NA	0.096	FAIL
Poly6	0.011	3.57	3.58	✓
Poly7	FAIL	NA	0.011	FAIL
Poly8	FAIL	NA	0.011	FAIL
Poly9	0.019	3.36	3.38	✓
Poly10	FAIL	NA	0.00	FAIL
Poly11	0.28	1.47	1.75	✓
Poly12	FAIL	NA	0.016	FAIL

Table 1: Name is the name of the benchmark; Infer time is the time taken by the infer phase of TpT in seconds. Validate time is the time in seconds taken by the validate phase of TpT to verify the candidate loop bounds. The fourth column shows the total time taken by TpT. The last column indicates the result of the analysis – whether TpT proved termination (✓), found a termination bug (×), or failed (FAIL).

Group	Total	TpT	O [4]	P [4]	PR [4]	T [4]	LR [7]	LTA [26]	LF [26]	CTA [26]
Oct	6	6	6	6	2	4	6	6	5	4
Driver	10	10	10	10	1	9	10	10	5	8
Poly	11	6	0	2	11	3	2	2	0	0
All	27	22	16	18	14	16	18	18	10	12

Table 2: Group is the benchmark group; the last row represents the sum of results over each of the three groups; Total is the number of benchmarks in the corresponding group; TpT is the number of benchmarks in each group on which TpT successfully completes its analysis; O is an Octagon based termination analysis [4]; P is a polyhedra based termination analysis [4]; PR is a script on top of [6]; T is Terminator [9]; LR is LinearRankTerm [7], an abstract-interpretation over an abstract domain which represents ranking functions; LTA is algorithmic-learning-based termination analysis [26]; LF is LoopFrog [39], a summary-based termination analysis; CTA is a compositional termination analysis [25].

it three times for each input value: one with nondeterminism replaced by *true*, one with nondeterminism replaced by *false*, and one with nondeterminism replaced by *rand*()%2. 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 particular test generation technique. On measuring the sensitivity to test data, we found that at most 300 randomly selected states, from the test data, were sufficient to obtain a sound bound for all our benchmarks.

For the micro-benchmarks, we used [36] for validation: it can handle non-linearities, consume candidate invariants, and requires tests. Unfortunately, on verification failure, [36] might not produce a reachable counterexample. Static analysis tools such as Yogi [16] do find reachable counterexamples but cannot handle non-linearities. For the micro-benchmarks, we run an increasing number of naively generated tests until [36] validates the inferred bounds.

The fifth column shows the results reported by TpT. A ✓ shows that TpT was able to prove the termination of all loops in the benchmark program. A × shows that TpT found a true non-termination bug. For such programs, TpT terminates with a suspect trace – it is important to note that TpT does not report a certificate of non-termination. However, our technique can be combined with non-termination provers in the spirit of [22] to derive a certificate of non-termination.

For all the benchmarks on which the infer phase succeeds in computing a bound, TpT is able to prove termination. For these cases, the free invariant obtained by linear regression was an actual invariant. However, the free invariant had to be strengthened by the validate phase to obtain an inductive invariant.

TpT required rounding for Oct3, Driver10, Poly3, and Poly6 and used the simple strategy described in Section 4.3. For the remaining benchmarks, the quadratic programming solver gave exact integral bounds. One major reason for why numerical techniques are not common in verification is

Name	LOC	#Loops	Infer time (sec)	Validate time (sec)	TpT time (sec)
kbfiltr	0.9K	2	0.001	8.8	8.8
diskperf	2.3K	4	0.001	41.8	41.8
fakemodem	3.1K	3	0.001	2841.7	2841.7
serenum	5.3K	17	0.04	2081.3	2081.3
flpydisk	6K	24	0.04	305.4	305.4
kbdclass	6.5K	16	0.05	1822.3	1822.4

Table 3: Performance of TpT on Windows drivers.

because of the difficulty in obtaining integral results. In the case of linear regression for inference of loop bounds, this concern is dispelled empirically. Rounding is usually not needed, and even when it is needed, it is quite simple. For the benchmarks `Driver10` and `Poly1`, TpT requires predicate based partitioning, and for the benchmark `Poly11`, TpT uses path based partitioning. It is interesting to note that `Poly1` can also be handled by path based partitioning.

TpT fails on the benchmarks `Poly4`, `Poly7`, `Poly8`, and `Poly12`. On preliminary examination, it seems like these benchmarks require more sophisticated partitioning of data. Development of learning algorithms for identifying these partitions is ongoing work. For instance, `Poly4` requires the data to be partitioned according to whether an input is positive or negative. TpT fails on `Poly10` as this benchmark has many assume statements and therefore it is hard to generate data by running the program and simultaneously satisfy all assume statements. Symbolic execution techniques are required for generating data for such benchmarks.

In terms of performance, the time taken by TpT is comparable with previous work. An exact comparison with the times reported by previous approaches is not meaningful (as we are running on newer hardware). However, we reproduce the number of benchmarks on which previous techniques are successful in Table 2 from their respective papers. From Table 2, it is important to note that TpT is able to successfully prove 15% more benchmarks in this suite than any previously known technique. One of the reasons for this is that regression is able to quickly guess sound bounds and invariants using a reasonable amount of data, as is evident from the times taken by the infer step. Moreover, unlike existing techniques, since the infer phase of TpT is driven by tests, it does not get confused by program text.

Finally, the results of TpT on Windows device drivers is reported in Table 3. We used an existing test suite that was available for these drivers (obtained by running the Yogi tool [3, 16]). The validate phase was implemented using the Yogi tool which is routinely used to check safety properties of Windows device drivers. As seen from the table, TpT is able to successfully prove termination for all these drivers. In contrast, Terminator [9] is unable to complete on the **serenum** and **flpydisk** device drivers (it runs out of memory when the memory limit is set to 1.8 GB). It is also interesting to note that the infer phase of TpT is very efficient and almost all the time taken by TpT is spent in validating the loop bounds. We believe that the infer phase of TpT which is driven by data from tests is almost independent of the size or complexity of the program.

7. RELATED WORK

Our work falls into the category of using machine learning for proving program properties [30]. There is a large body of work on termination and bounds analysis. We draw a comparison with techniques that are most closely related to TpT.

Regression has also been used to determine the execution time of programs [23], for empirical computational complexity [14], and profiling [40]. These approaches have no soundness guarantees. We have shown that regression can be used to not only obtain sound bounds, sufficient for proving termination, but also for discovering invariants required for proving the bounds. In contrast to these approaches, we use a constrained variant of regression that is efficiently solvable by quadratic programming.

There is a lot of work that addresses the problem of mining useful information from tests [12, 24]. Deriving invariants from tests for the purpose of proving safety properties has been pioneered by the Daikon approach [13]. In contrast, TpT is a novel testing based approach for proving program termination. In safety checking, a candidate invariant is refuted by a counter-example state which does not satisfy the candidate [35–37]. Interestingly, for TpT the counter-example state is not very relevant, since termination is not a safety property. What is relevant for TpT is the number of times a loop body iterates for a given counter-example.

Podelski and Rybalchenko [31] describe a complete procedure for proving termination for a restricted class of programs. Cousot [10], Gulwani et al. [19], and Bradley et al. [5] compute ranking functions by using a template and solving constraints encoding the program. TpT, on the other hand, operates on data generated from tests, and the constraints which verify that a given bound is an actual bound in the validate phase are much simpler to solve than the constraints used to synthesize ranking functions.

Bradley et al. [6] developed termination techniques based on the lexicographic polyranking principle. These techniques can prove termination of all the tricky loops in the `Poly` benchmarks described in Section 6. These techniques were originally developed for transition systems, and therefore they find it hard to deal with complex control flow. In contrast, the infer phase of TpT does not even look at the program and in the validate phase, the safety checker just encodes control flow as a constraint which is subsequently handed off to an SMT solver. Since TpT does not rely on predicate abstraction for inferring termination arguments, it is also able to outperform techniques based on algorithmic learning [26].

Techniques for handling termination of non-linear programs are based on a restricted program syntax because they want their termination argument to be sound by construction. For instance, Babic et al. [1] can prove the termination of non-linear programs with a restricted syntax by finding "regions of guaranteed divergence". Due to their restricted syntax, they cannot prove the termination of the program in Figure 8. In contrast, TpT does not require its infer phase to be sound. This allows us to use predictive machine learning algorithms which work well in practice for the synthesis of bounds. TpT recovers soundness via the validate phase. Gulavani et al. [15] adopt a similar approach and try to find a bound for an instrumented loop counter by abstract interpretation. Using their ideas, we can extend our technique to infer loop bound expressions which can contain logarithms, exponentials, and square roots. The idea is to just add a column to the data matrix for every feature that can occur in the bound and provide enough axioms to the checker to reason about the features.

Proofs using transition invariants [25, 32] require a well chosen partitioning. Obtaining this partitioning is a well known hard problem and several heuristics exist. Terminator [9] is based on transition invariants and performs partitioning iteratively. It will be useful to see if abstraction refinement techniques like those employed by Terminator can help us obtain a good partitioning. [4, 7, 39] are abstract interpretation based approaches for termination. These approaches fail when non-linear computations are involved. We note that [4] leaves handling disjunctions as an open problem and [7] handles disjunctions by enumerating all paths in a loop. TpT requires partitioning when no bounds exist. In most loops occurring in practice (as attested by our experiments), the loops do have linear or polynomial bounds.

Together with proving termination, it seems important to produce certificates of non-termination for potentially non-terminating programs. We are exploring whether our technique can be combined with [20] in the spirit of [22].

Our technique is related to bounds analyses of Gulwani et al. [17, 18]. These approaches find it difficult to handle non-linearities. As a consequence, they instrument the program with multiple counters with the hope that all counters will have linear bounds. In contrast, we can obtain non-linear bounds. It is important to note that Gulwani et al. can find more precise bounds than TpT as their bounds can contain richer operators like *max*. For the program in Figure 1, Gulwani et al. can compute the bound $max(a, b)$. On the other hand, TpT infers a looser bound $a + b - 2$. Since our goal is to obtain bounds for the purpose of proving termination, these loose bounds suffice. That said, since we are using linear regression, our bounds are not very loose. But since they are restricted to linear or polynomial expressions, they are not as tight.

When the number of iterations of a loop is a deterministic function of the input, then [38] can use dynamic information and polynomial interpolation to compute a loop bound which is subsequently validated using a proof assistant. However, many of our examples have internal non-determinism and therefore an interpolating polynomial does not exist (while a loop bound still exists).

8. CONCLUSION

In this paper, we have presented a novel algorithm TpT for proving termination of programs using information derived from tests. Programs usually have test suites associated with them, and we are able to learn loop bounds automatically from these test suites using a modification of linear regression. We have also shown how to extend linear regression so as to infer polynomial loop bounds.

Our empirical results show that TpT significantly improves the state-of-the-art in termination analysis – on micro benchmarks, TpT is able to prove termination on 15% more benchmarks than any previously known technique. Our results over Windows device drivers demonstrates TpT's ability to analyze and scale to real world applications.

There are number of interesting directions for future work: **Automatic partitioning:** As seen in Section 6, TpT fails to prove termination on four benchmarks all of which require more sophisticated partitioning of data. It would be interesting to see if clustering algorithms in machine learning can be combined with regression to infer disjunctive loop bounds systematically.

Other instantiations of TpT: It would also be interesting to extend TpT for proving liveness properties as well as total correctness of programs. We believe that these would require a tighter integration of TpT with a safety checker.

9. ACKNOWLEDGEMENTS

We thank Saurabh Gupta, Bharath Hariharan, Andreas Podelski and Sriram Rajamani for many insightful comments and suggestions.

10. REFERENCES

[1] D. Babic, A. J. Hu, Z. Rakamaric, and B. Cook. Proving termination by divergence. In *Software Engineering and Formal Methods (SEFM)*, pages 93–102, 2007.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2005.

[3] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14, 2008.

[4] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *Principles of Programming Languages (POPL)*, pages 211–224, 2007.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *Computer Aided Verification (CAV)*, pages 491–504, 2005.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *International Colloquium on Automata, Logic and Programming (ICALP)*, pages 1349–1361, 2005.

[7] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *European Symposium on Programming (ESOP)*, pages 148–162, 2008.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169, 2000.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Programming Language Design and Implementation (PLDI)*, pages 415–426, 2006.

[10] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 1–24, 2005.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[12] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2):243–257, 2012.

[13] 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*, 69(1-3):35–45, 2007.

[14] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Foundations of Software Engineering (FSE)*, pages 395–404, 2007.

[15] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification (CAV)*, pages 370–384, 2008.

[16] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *Foundations of Software Engineering (FSE)*, pages 117–127, 2006.

[17] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Programming Languages Design and Implementation (PLDI)*, pages 375–385, 2009.

[18] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[19] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation (PLDI)*, pages 281–292, 2008.

[20] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *Principles of Programming Languages (POPL)*, pages 147–158, 2008.

[21] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 262–276, 2009.

[22] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *Static Analysis Symposium (SAS)*, pages 304–319, 2010.

[23] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Neural Information Processing Systems (NIPS)*, pages 883–891, 2010.

[24] A. Komuravelli, C. S. Pasareanu, and E. M. Clarke. Learning probabilistic systems from tree samples. In *Logic in Computer Science (LICS)*, pages 441–450, 2012.

[25] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *Computer Aided Verification (CAV)*, pages 89–103, 2010.

[26] W. Lee, B.-Y. Wang, and K. Yi. Termination analysis with algorithmic learning. In *Computer Aided Verification (CAV)*, pages 88–104, 2012.

[27] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[28] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering (ICSE)*, 2012.

[29] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., 1982.

[30] C. S. Pasareanu and M. G. Bobaru. Learning techniques for software verification and validation. In *International Symposium on Leaveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 505–507, 2012.

[31] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 239–251, 2004.

[32] A. Podelski and A. Rybalchenko. Transition invariants. In *Logic in Computer Science (LICS)*, pages 32–41, 2004.

[33] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., 2005.

[34] R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification (CAV)*, pages 703–719, 2011.

[35] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *European Symposium on Programming (ESOP)*, pages 574–592, 2013.

[36] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Program verification as learning geometric concepts. In *Static Analysis Symposium (SAS)*, 2013.

[37] R. Sharma, A. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification (CAV)*, pages 71–87, 2012.

[38] O. Shkaravska, R. Kersten, and M. C. J. D. van Eekelen. Test-based inference of polynomial loop-bound functions. In *Principles and Practice of Programming in Java (PPPJ)*, pages 99–108, 2010.

[39] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop summarization and termination analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 81–95, 2011.

[40] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Programming Language Design and Implementation (PLDI)*, pages 67–76, 2012.