NASA/TM-2011-215962

Symbolic Execution Enhanced System Testing

Misty Davies
Ames Research Center
Moffett Field, California

Corina Pasareanu
Carnegie Mellon University
Ames Research Center
Moffett Field, California

Vishwanath Raman
Carnegie Mellon University
Ames Research Center
Moffett Field, California

March 2011

NASA STI Program... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collects papers from scientific and technical conferences, synopsia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2011-215962

Symbolic Execution Enhanced System Testing

Misty Davies
Ames Research Center
Moffett Field, California

Corina Pasareanu
Carnegie Mellon University
Ames Research Center
Moffett Field, California

Vishwanath Raman
Carnegie Mellon University
Ames Research Center
Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California, 94035-1000

March 2011

Available from:

NASA Center for Aerospace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

National Technical Information Service
5285 Port Royal Road
Springfield. VA 22161
703-487-4650

Symbolic Execution Enhanced System Testing *

Misty Davies
NASA Ames Research Center
Moffett Field, CA 94035
misty.d.davies@nasa.gov

Corina Păsăreanu
Carnegie Mellon University
Moffett Field, CA 94035
corina.s.pasareanu@nasa.gov

Vishwanath Raman
Carnegie Mellon University
Moffett Field, CA 94035
vishwa.raman@west.cmu.edu

ABSTRACT

We describe a novel testing technique that uses the information computed by a symbolic execution of a program unit to guide the generation of inputs to the system containing the unit, in such a way that the unit's, and hence the system's, coverage is increased. The symbolic execution is performed at run-time, along program paths obtained by system level simulations. Data mining techniques are used to obtain a first approximation of the system level input constraints that influence the satisfaction of the unit level constraints computed by the symbolic execution of the unit. Function fitting is used to incrementally approximate the behavior of the unit's calling context. Finally, constraint solving for the unit level constraints, together with the learned approximating functions and system level constraints, are used to predict the system level inputs that uncover new code regions in the unit under analysis. We demonstrate the effectiveness of our technique on two illustrative examples and on a realistic example from the NASA domain.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; G.1.2 [**Mathematics of Computing**]: Approximation

General Terms

Coverage, Testing, Learning

Keywords

System Testing, Symbolic Methods, Simulation, Machine Learning, Function Fitting

1. INTRODUCTION

Modern software, and in particular flight control software like that written at NASA, needs to be highly reliable and hence thoroughly tested. System level Monte Carlo simulations are typically used for testing NASA software. Such system level "black-box" simulations have the advantage that they are easy to set up, since the user only needs to specify the ranges for the system level inputs, but they provide few guarantees in terms of testing coverage. Furthermore, system level simulation may be quite expensive, as the system under analysis includes not only the flight software, but also various models of the physical environment and of the space vehicle hardware. For example, a run using NASA's ANTARES simulator [1] may take hours to complete.

Recently, a new set of techniques [22, 26, 15], based on symbolic execution [19], have emerged, for generating test cases that achieve high code coverage. Symbolic execution, and its variant, concolic execution, are "white-box" as they collect constraints based on the *internal* code structure. The collected constraints are solved systematically to obtain inputs that exercise all the paths through the code. However, the techniques are expensive, both in terms of program paths to explore and in the number of complex constraints to be solved. Therefore, they can be used effectively for testing individual software units (inside a system) but can hardly scale to the whole system. However, it is often the case that the unit level inputs are constrained by the unit's system level calling context. To obtain realistic test cases, such constraints need to be encoded explicitly, which would require non-trivial manual effort from developers.

The goal of the work reported here is to study the synergy between "black-box" system simulation and "white-box" unit symbolic execution to overcome their weaknesses. We propose an iterative procedure that uses the information computed by a symbolic execution of a unit to *guide*, via machine learning techniques, the generation of new system level inputs that increase the coverage of the unit, and hence of the system containing the unit. Thus, our proposed approach improves upon system level testing by increasing the obtained coverage with a reduced number of test cases, and hence with a reduced cost. Furthermore, it enables a modular, more scalable, unit level analysis under *realistic* contexts, since symbolic execution is performed only along the program paths obtained via simulation.

Specifically, we use data mining techniques (i.e. treatment learning [21]) to obtain a first approximation of the system level input constraints that influence the satisfaction of the unit level constraints computed by the symbolic execution of the unit. Function fitting is performed to incrementally approximate the behavior of the unit's calling context. Finally, the unit level constraints are solved with off-the shelf constraint solvers and the approximations together with the system level constraints are used to guide the generation of new system level inputs towards executing uncovered code regions in the unit under analysis.

We have implemented the techniques for function fitting, treatment learning and symbolic execution in the context of the analysis of C programs that perform complex, non-linear mathematical computations. We report here on the application of our approach on several non-trivial examples from the NASA domain.

2. BACKGROUND

In this section we provide background relevant to the rest of the paper. We introduce a program model followed by concolic exe-

*This research was conducted at Carnegie Mellon University and the Ames Research Center under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government

cution, a technique that combines symbolic and concrete program execution to enhance path coverage. We then introduce a machine learning technique called treatment learning [21] followed by a brief description of function fitting. We use these techniques together with Monte Carlo simulations for system level testing.

2.1 A Program Model

We define a program as the tuple $P = (I, A, C)$, where I is a set of input parameters, A is a set of assignment statements and C is a set of conditional statements. We assume that the elements of I are of *basic types*, which we define to be a type from the set $\{int, short, unsigned\, int, char, float, double, enum\}$, with each element $i \in I$ taking values from a domain D_i based on its type. Further, we assume that all assignment and conditional statements refer to elements in I. We define the set of all executions of the program P as $R(P) \subseteq (A \cup C)^*$; a set of finite sequences of assignments and conditional statements visited over all possible values of the parameters in I. An assignment over the parameters in I, denoted $\vec{I}$, associates every element $i \in I$ to a value in D_i. Given an assignment $\vec{I}$, we assume that all executions of the program visits exactly the same finite sequence of assignments and conditional statements; the programs are deterministic.

2.2 Concolic Execution

Concolic execution, introduced in [15, 25], is a technique that combines concrete and symbolic program execution to increase path coverage. The concrete execution of a program $P = (I, A, C)$, given an assignment $\vec{I}$ over I, leads to a unique path in P characterized by the conjunction of the conditional expressions that evaluated to true. Using symbolic names for the parameters in I, every conditional expression can be represented symbolically so that the path taken in P is uniquely characterized by the conjunction of these symbolic terms; such a conjunction of symbolic terms is called a Path Constraint (PC). Every PC characterizes a unique path taken in P. Given a PC, by exhaustively negating terms in the constraint we can generate new path constraints for paths not taken during concrete execution of P. When each new path constraint is submitted to a constraint solver, we have that the constraint is either satisfiable or unsatisfiable. If the constraint is satisfiable, then by using the satisfiable assignment over I, returned by the constraint solver, we can guide concrete program execution to visit the path characterized by the constraint. If the constraint is unsatisfiable, then the path cannot be taken by any concrete program execution. Therefore, given a program $P = (I, A, C)$, concolic execution attempts to exhaustively cover all paths that can be taken by a program, by selecting assignments over the input parameters I. We use the code fragment in Program 1 to explain the technique.

Consider the function $swap$ in Program 1. It swaps the contents of the pointer parameters x and y without using temporaries. We are interested in exploring all possible paths that can be taken by Program 1 and in particular, we would like to check if the assert in Line 6 is ever executed; if we assume that the values of x and y when we execute the function are such that $*x > *y$, then the second condition is always $false$. To execute the function symbolically, we first associate the parameters $*x$ and $*y$ with symbolic names x and y. The PC is initially set to $true$. When we encounter the first conditional statement in Line 1, since $(*x > *y)$, we conjoin the symbolic expression $(x > y)$ to PC. When we encounter an assignment, we bind the symbolic expression representing the right hand side to the symbolic variable on the left hand side. For instance, after executing Line 2, $*x$ is bound to the symbolic expression $x + y$. At the point at which we test the second condition in Line 5, it is easy to see that $*x$ is bound to the symbolic expression $((x + y) - ((x + y) - y))$ and $*y$ is bound to the symbolic expression $((x + y) - y)$. Since the second conditional statement always evaluates to $false$, it is easy to see that if we start with $(*x > *y)$, the value of PC when the function returns will be $true \wedge (x > y) \wedge (y \leq x)$. To check if the $then$ branch in Line 5 is ever taken, we negate the last term in the constraint to yield $true \wedge (x > y) \wedge (y > x)$, which when submitted to a constraint solver is unsatisfiable. This implies the assert in Line 6 is never executed. This exhaustive exploration technique has been used effectively to explore paths in C programs.

Program 1 *Swap without temporaries*

```
   void swap(int* x, int* y)
   {
1    if (*x > *y) {
2      *x = *x + *y;
3      *y = *x - *y;
4      *x = *x - *y;
5      if (*x > *y)
6        assert(false); // should not happen
     }
   }
```

2.3 Treatment Learning

Treatment learning [21] is a machine learning technique that finds the minimal difference between two sets. It is generally used as a crude optimization technique or as a sensitivity analysis technique. The algorithmic details of the technique can be found in [21] and [14]. The goal of treatment learning is to determine a *small number* of controllable inputs and ranges that are most likely to lead to some output. These inputs and their ranges are known as a *treatment* or a rule. This is in contrast to many other association rule learners [4, 23, 7] that potentially find more accurate rules at a cost of greater complexity and time [18, 20]. For this work, we are using TAR3 as our treatment learner.

TAR3 is faster than other optimization methods for finding association rules and handles relationships involving both continuous and discrete variables [14]. Given a data set and a partition of that set into a set of desired data points and a set of all remaining points, TAR3 looks for rules that are subsets of input parameters and their ranges, that maximize the likelihood of seeing points in the desired set. The rules returned by TAR3 are subspaces of intersecting hyperplanes. Graphically, TAR3 gives rules that are bounded by boxes around promising data ranges. As an example, refer to Figure 1. Each asterisk in this plot represents a valuation in the input space. In the example, our set of desired data points are the asterisks with small boxes around them. The ellipse represents one set of desired data points. Given the set of desired points and all other points, TAR3 returns a rule, which is a set of parameters and their ranges, that increases the likelihood that we will see data like those that we desire. The dashed rectangle around the three asterisks at the top of the plot is the rule that TAR3 learned for the data set in the figure.

2.4 Function Fitting

Function fitting is used to determine a predictive relationship between outputs and inputs, given some number of measurements. Function fitting is most likely to be successful between one output variable and a small number of input variables. We use function fitting to approximate a relationship between the unit level inputs given to us during concolic execution and their associated system level input variables. We find the functions using a discrete least squares approximation. This has the advantage of being amenable to well-studied numerical linear algebra techniques [28, 27] and of

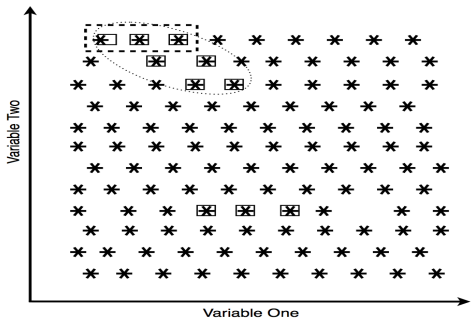

Figure 1: *A plot showing measurements taken for two parameters. Each asterisk represents a measurement. If an asterisk is boxed, that data point is one that we would like TAR3 to find a rule or* treatment *for. The dashed rectangle is a treatment. The dotted ellipse shows a potential rule that was not given by the treatment learner.*

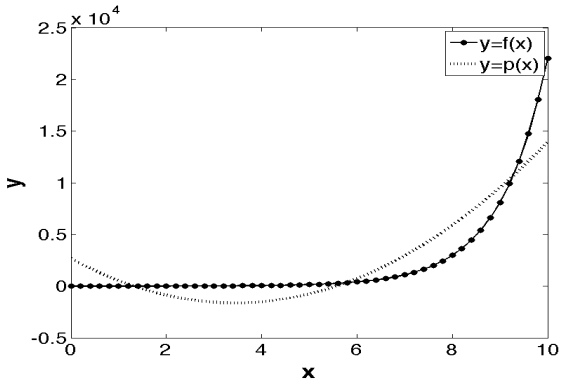

Figure 2: *A plot showing $y = f(x)$ where $f(x) = e^x$ on the domain 0 to 10, along with its best quadratic least squares approximation $p(x)$. The dots along the curve $y = f(x)$ show where the measurements were taken, and $y = p(x)$ is the quadratic curve that minimizes the sum of the distances between $f(x)$ and $p(x)$ at the measurements x.*

being less sensitive to outliers than many competing techniques [5].

Consider a function $y(x)$ that we would like to approximate by a function $p(x)$. A least squares solution finds some number of constant values c_i, for $i = \{1, 2, 3, \ldots\}$, that minimize the total Euclidean distance between $p(x)$ and $y(x)$ at the measurements x. This sum of distances is called the *residual*. Figure 2 shows the least squares solution $p(x) = c_1x^2 + c_2x + c_3$ for the function $y(x) = e^x$ and measurements taken at x between 0 and 10 with measurements 0.2 apart. We could use any approximating functions, but choose to use polynomials as they have several advantageous properties. If we assume that the relationships we are trying to approximate are *smooth*, where by *smooth* we mean Lipschitz continuous, then we can find a polynomial approximation that is arbitrarily close to our desired function by the Weierstrass Approximation Theorem [3]. A function that is not smooth along its entire domain may be *locally smooth*, or smooth along some subinterval of the domain. A polynomial constructed on this subset is known as a *piecewise-polynomial approximation*. For a smooth function, shrinking the subinterval on which the approximation is made allows for arbitrarily close approximations with low-order polynomials [24].

Table 1: *A table showing the number of measurements needed in order to fit polynomial approximations of up to cubic order for one output variable with up to 4 input variables.*

	Threshold values by number of variables			
Polynomial fit	1	2	3	4
linear	2	3	4	5
quadratic	3	6	10	15
cubic	4	10	20	35

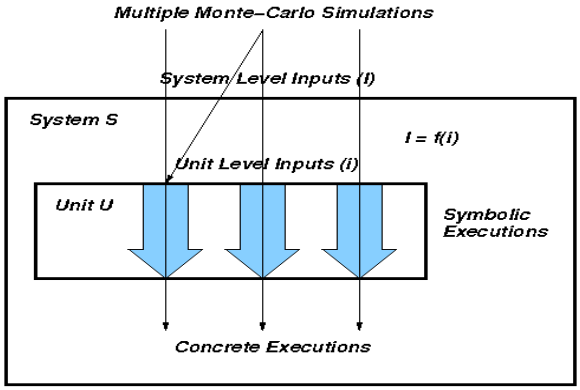

Figure 3: *A system S with inputs I and an embedded unit U with inputs i.*

In this work, we fit one output variable i to a subset of the input variables I. We must solve for every constant c in front of each term. For every unknown c, we need at least one equation, which means we need at least one measurement. Having exactly the same number of measurements as unknown constants c_i means that we can find an exact solution. If we have more measurements than unknowns, then the problem is *overdetermined* and the solution will be the least squares solution. Increasing the number of variables I increases the number of measurements needed, as does increasing the order of the approximating polynomial, as shown in Table 1. We refer to the minimum number of measurements needed for a given number of variables as the *Threshold*.

3. APPROACH

In this paper we are concerned with increasing path coverage during system-level testing. While concolic execution has been shown effective in covering paths in programs, the technique fails when the program being tested becomes too large. On the other hand, Monte Carlo simulations may not cover interesting corner cases even with very large sets of random assignments over system inputs. The systems we encounter are typically large with large non-linear fragments. We therefore find that during system level testing, many paths leading potentially to unsafe program states are left uncovered. Our approach aims to merge the results of learning-based directed heuristic testing with concolic execution to increase system level path coverage. Figure 3 captures the systems we are interested in testing.

In the figure, we show a System Under Test (SUT) that consists of one or more white-box program units. Each white-box unit is an interesting code fragment that lends itself to concolic testing; typically these are code fragments with linear constraints and of small enough size that they can be completely covered using concolic execution. In Figure 3, S is a system with input parameters I containing a unit $U = (i, A, C)$ with unit level parameters i. A

Program 2 *Prototype Linear Example*

```
int g1 = 1, g2 = 2;
int System(int I1, int I2)
{
  if (I1 > 0) g1 = I2; else g1 = -I2;
  g2 = I1 + 3;
  Unit(I2, I1);
}
int Unit(int i1, int i2)
{
  if(i1 > 0) {
    i2 = g2;
    if(i2 > 0) return 0; else return 1;
  } else {
    i2 = g1 + 3;
    if(i2 > 0) return 2; else return 3;
  }
}
```

subset of the behavior of the system is then some function f of the system level parameters I that computes values for the inputs i of the unit U. Let $c \in C$ denote some conditional statement in U that was not covered during system level testing. Let $Cons(c)$ denote the unit level constraint, over parameters in i, associated with statement c; for example if $i = \{v, w\}$, a constraint could be $(v > w)$. Since concolic execution of U excludes the system that instantiates U, it generates an over-approximation of the set of paths that can be covered during system level testing. By the same token, paths that are unreachable in U remain unreachable in S; a path unreachable in the most liberal environment for U remains unreachable in the restricted environment provided by S. If $Cons(c)$ is satisfiable, then a satisfying valuation $\vec{i}$ will enable us to cover statement c at the unit level. Our approach is then to generate assignments over the system level parameters I, given the satisfying valuation $\vec{i}$ at the unit level, that can cover statement c during system level testing. To solve this, we assume f is invertible and take $I = f(i)$, for the unknown function f, and use machine learning techniques to approximate f. Once we have an approximation for f, we attempt to cover statement c by composing a system level test vector using $I = f(i)$, evaluated for the valuation $\vec{i}$. We describe our approach in detail in the next section.

4. TESTING ALGORITHMS

As a running example, consider the program in Figure 2. There are two system level integer valued parameters $I1$ and $I2$. Further since globals may be referred in either System or Unit, we consider the two integer valued global variables $g1$ and $g2$ as parameters to both System and Unit. The unit level inputs are therefore $i1$, $i2$, $g1$ and $g2$. While the program is linear, we will use this example to illustrate key concepts in our approach.

4.1 Constraints Trees

We use concolic execution on the program fragment called *Unit*. The set of input parameters is $i = \{i1, i2, g1, g2\}$. Since Unit is small and linear we achieve full path coverage using concolic execution. Given an assignment over the parameters in i, each path taken in Unit is precisely characterized by the path constraint PC when Unit returns. All such path constraints, accumulated over all executions of Unit, are stored in a tree which we call the *Constraints Tree*. The constraints tree reflects the set of all paths that were taken by all executions of a program unit.

In Figure 4 we have a constraints tree for Unit after some initial testing. The parameters section contains the set of parameters

```
  [Parameters]
2 i1
3 g2
4 g1
  [Tree]
7   (i1 > 0) (C, 0, 248, 0)
8     (g2 > 0) (C, 0, 250, 0)
9     (g2 <= 0) (S, 0, 250, 1)
10  (i1 <= 0) (C, 0, 248, 1)
11    ((g1 + 3) > 0) (C, 0, 261, 0)
12    ((g1 + 3) <= 0) (S, 0, 261, 1)
```

Figure 4: *The constraints tree after some rounds of initial testing*

that are involved in the constraints in the tree. The tree section contains a textual representation of the constraints tree. The number of constraints in the tree is equal to the number of leaves in the tree and each constraint is a conjunction of the terms encountered along the parent hierarchy of each leaf. Therefore, given the tree in Fig 4, the set of constraints are, $(i1 > 0) \wedge (g2 > 0)$, $(i1 \leq 0) \wedge (g2 \leq 0)$, $(i1 \leq 0) \wedge ((g1 + 3) > 0)$ and $(i1 \leq 0) \wedge ((g1 + 3) \leq 0)$. Of these constraints, $(i1 > 0) \wedge (g2 > 0)$ and $(i1 \leq 0) \wedge ((g1+3) > 0)$ were covered during our initial testing. The text within parentheses after each term is annotated with a string of the form $([C|U|S|?], \mathbb{N}, \mathbb{N}, \mathbb{N})$, where the first character is "C" if the term was true and hence the constraint leading up to it was *Covered* during a run, "U" if the constraint leading up to it is *Unsatisfiable*, "S" if the constraint leading up to it is *Satisfiable* and "?" if the constraint leading up to it is unknown. By using models returned by a constraint solver for the two satisfiable constraints, we obtain assignments over the parameters in i that completely cover all paths in Unit.

4.2 Observations

We consider a system S, with system level parameters I, and a unit U within S, with unit level parameters i. We assume that the unit can be analyzed using concolic execution. Let T be a constraints tree extracted by monitoring U during system level testing. Consider nodes in T that are satisfiable at the unit level but not covered by system level testing. Since we cannot use concolic execution at the system level, our coverage algorithm attempts to cover all nodes that are satisfiable at the unit level using a combination of concolic execution, treatment learning and function fitting. For a node $n \in T$ we take $Cons(n)$ as the unit level constraint that leads to n and that when satisfiable will cover n at the unit level. In order to present our coverage algorithm, we first present the following observations that are used by the algorithm.

Consider a path $\sigma = n_1, n_2, \ldots, n_k$ in a constraints tree T such that all nodes $n_1, n_2, \ldots, n_k$ are covered by system level testing. Since the nodes were covered, there exist vectors at the system and unit level that are witnesses to cover the nodes in σ, i.e., for a set of system level vectors V_i that cover node n_i in σ, there exists a corresponding set of unit level vectors v_i that covered n_i. We then have the following properties of these witnesses.

OBSERVATION 1. **(Monotonicity of Witnesses)** *For a constraints tree T and a path $\sigma = n_1, n_2, \ldots, n_k$ of nodes in T, such that $n_1, n_2, \ldots, n_k$ are covered with witness sets $V_1, V_2, \ldots, V_k$ at the system level and corresponding sets $v_1, v_2, \ldots, v_k$ at the unit level, we have, $V_1 \supseteq V_2 \supseteq \ldots \supseteq V_k$ and $v_1 \supseteq v_2 \supseteq \ldots \supseteq v_k$.*

Monotonicity of Witnesses follows easily by noting that $Cons(n_k) \Rightarrow Cons(n_{k-1}) \Rightarrow \ldots \Rightarrow Cons(n_1)$ for the constraints of nodes in σ.

OBSERVATION 2. **(Sufficiency of Witnesses)** *For a constraints tree T and a path $\sigma = n_1, n_2, \ldots, n_k$ of nodes in T, such that $n_1, n_2, \ldots, n_k$ are covered with witness sets $V_1, V_2, \ldots, V_k$*

at the system level and corresponding sets $v_1, v_2, \ldots, v_k$ at the unit level, let $|V_j| \geq Threshold$ such that for all $i \in [1,k]$ with $|V_i| \geq Threshold$, we have $|V_j| \leq |V_i|$. If the relation between the vectors in V_j and v_j is not smooth for function fitting, then it is the case that for all $i \leq j$, the relation between V_i and v_i is also not smooth for function fitting.

Consider a constraints tree T and a path $\sigma = n_1, n_2, \ldots, n_k$ in T, such that all nodes that precede n_k are covered during system level testing, but node n_k is not covered. Since we cannot use concolic execution on the entire system S, we have that $Cons(n_k)$ is the finest symbolic path constraint, such that when $Cons(n_k)$ is satisfiable, using the assignment that satisfies $Cons(n_k)$, we can cover n_k at the unit level. We take $Term(n_k)$ as the term corresponding to the node n_k and $Parent(n)$ as the parent of a node n in σ. Given a constraint C, we take $Vars(C)$ as the set of parameters that appear in the terms of constraint C. The path constraint $Cons(n_k)$ is then $Term(n_1) \wedge Term(n_2) \wedge \ldots \wedge Term(n_k)$. As we cannot extend the path constraint to the system level, we would like to learn the system level behavior as a function f, such that $I = f(Vars(Cons(n_k))$, via function fitting. Given $Cons(n_k)$ is satisfiable, we can then use f to find a system level vector that covers n_k using the satisfying assignment over $Vars(Cons(n_k))$ as returned by a constraint solver for $Cons(n_k)$. But the caveat in this approach is that function fitting is in general difficult over large data sets due to both the number of parameters involved and due to the presence of discontinuities. We tackle this problem as follows:

- We attempt function fitting for a constraint C, starting at $Term(n_k)$, progressively conjoining terms $Term(n_i)$ for $i = k-1, k-2, \ldots, 1$, stopping as soon as we succeed in finding a smooth function. This reduces the number of unit level parameters we consider and by the sufficiency of witnesses uses the smallest number of data points needed to fit a smooth function.
- We reduce the number of system level parameters for function fitting by using treatment learning. For a constraint C, we use the data seen during system level testing to find the subset $I_n \subseteq I$ of system level parameters that have the highest likelihood of affecting the values of the unit level parameters in $Vars(C)$.

This reduces the number of parameters for function fitting and avoids discontinuities by focusing on finding functions that are locally smooth. For all terms in $Cons(n_k)$ that are not considered in a given iteration of function fitting, i.e., terms in $Cons(n_k)$ but not in C, we use treatment learning to find assignments for system level parameters that satisfy those terms. By the monotonicity of witnesses, we have more data points to cover these terms than those that cover $Cons(n_k)$, thus increasing the likelihood of finding good treatments.

4.3 Algorithm

We now describe $Cover$, our coverage algorithm presented in Algorithm 1. The algorithm works as follows:

1. *Lines 2–4*. We perform n-factor combinatorial Monte Carlo (MC) simulations by picking values over a space sp; a d-dimensional space for d input parameters constrained by their data types. In contrast to traditional random MC, n-factor MC generates test cases such that every possible combination of input parameters equal to size n appears at least once in the test suite [9]. For every system level vector a, we monitor the unit and capture the unit level vector b together with the path constraint for the path taken within the unit. At the end of this step, the set of path constraints that summarize all the execution paths that were taken in the unit are available in a constraints tree T; the system and unit level vectors are stored in sets V and v.
2. *Lines 8–10*. We traverse the nodes in T in breadth first order. We run the treatment learner for each node $n \in T$ that was covered as long as its sibling is also covered; this is necessary as treatment learning is efficient at identifying system level parameters and their ranges that have the highest likelihood of reaching n, in the presence of data points that differentiate n from its sibling. This step gives us a subset I_n of system level parameters and their ranges in R_n that have the highest likelihood of reaching n.
3. *Lines 12–15*. For each node $n \in T$ that is satisfiable, but that was not covered by our MC simulations, we store the assignment, $\vec{i}$, over i that satisfies the path constraint $Cons(n)$. We then start with a constraint C set to $Term(n)$ with the expectation that we will progressively strengthen C as we attempt to find a system level vector to cover n. Since we want to fit a function that maps system level inputs to the unit level inputs, we keep track of the parameters in C in i_n and the restriction of $\vec{i}$ to the parameters in i_n in $\vec{i_n}$. We call the function $ComputeMap$ to find a function f_n such that $I_n = f_n(i_n)$ using function fitting.
4. *Lines 17–18*. We iterate over all nodes $n \in T$ that are satisfiable at the unit level but were not covered during system

Algorithm 1: $Cover(S, U)$

input : System S with inputs I with $d = |I|$, unit U with inputs i

```
1  sp ← IR^d;
2  Perform n-factor combinatorial MC simulations over space sp;
3  (V, v) ← {(a, b) | a is a system level vector and b is the corresponding monitored unit level vector};
4  T ← (PC from U);
5  repeat
6      T' ← T;
       // Do BFS on T
7      for (node n ∈ T using BFS) do
8          if (n and n's sibling are covered) then
               // Since we have contrasting
               // data, learn a treatment
9              V' ← {a ∈ V | a covers n} and
               V'' ← V \ V';
10             (I_n, R_n, _) ← RunTAR3(I, V, V', V'');
11         else
12             if (n is satisfiable but not covered) then
                   // Compute f_n such that
                   //  I_n = f_n(i_n)
13                 i⃗ ← model for Cons(n);
14                 C ← Term(n);
15                 (I_n, i_n, f_n) ←
                   ComputeMap(C, I, V, v, n, Parent(n), i⃗);
       // Build new testcases
16     for (n ∈ T satisfiable but not covered) do
17         Run S with a consistent valuation using f_n(i_n⃗)
           and ∀j ∈ I using R_j from 10;
18         T' ← T' ∪ (PC from U);
19     T ← T';
20 until (T has no unprocessed nodes);
```

Algorithm 2: $ComputeMap(C, I, V, v, n, n', \vec{i})$

input : Constraint C such that $Cons(n_k) \Rightarrow C$, system inputs I, system level vectors V, unit level vectors v, a node n that we want to cover, a node n' that is in the parent hierarchy of n and a model $\vec{i}$ for $Cons(n)$

output: (I_n, i_n, f_n) where $I_n = f_n(i_n)$ and $i_n = Vars(C)$)

```
1  i_n ← Vars(C);
2  i_n⃗ ← restriction of i⃗ to i_n;
   // Find a subset of I for function
   // fitting
3  V' ← {a ∈ V | a is in 20% of points closest to Cons(n)}
   and V'' ← V \ V';
4  (I_n, R_n, smooth) ← RunTAR3(I, V, V', V'');
5  if (smooth) then
6  |  Build map I_n = f_n(i_n);
7  else
   |  // Strengthen constraint and try
   |  // again
8  |  if (n' exists) then
9  |  |  C ← C ∧ Term(n');
10 |  |  (I_n, i_n, f_n) ←
   |  |  ComputeMap(C, I, V, v, n, parent(n'), i⃗);
11 |  else
   |  |  // If we have no smooth relation
   |  |  // between I_n and i_n, then
   |  |  // walk up the parent of n
   |  |  // and pick a node with Threshold
   |  |  // points and attempt a linear fit
12 |  |  n'' ← n;
13 |  |  while (Parent(n'') exists) do
14 |  |  |  C ← C ∧ Term(Parent(n''));
15 |  |  |  n'' ← Parent(n'');
16 |  |  |  V' ← {a ∈ V | a covers n''};
17 |  |  |  if (|V'| ≥ Threshold) then
18 |  |  |  |  break;
19 |  |  V'' ← V \ V';
20 |  |  (I_n, R_n, _) ← RunTAR3(I, V, V', V'');
21 |  |  i_n ← Vars(C);
22 |  |  Build map I_n = f_n(i_n);
```

level testing. For each such node we run a system level test by composing a system level vector as follows: (a) take $I_n = f_n(\vec{i_n})$ such that it is consistent with the ranges R_j for all $j \in I_n$ as returned by the treatment learner in Line 10 and (b) for all other system level parameters $j \in I \setminus I_n$, pick a value from the ranges R_j returned by the treatment learner in Line 10.

The function fitting algorithm $ComputeMap$, shown in Algorithm 2, works as follows:

1. *Lines 1–4* We first compute the parameters i_n that occur in the constraint C and the restriction of the model $\vec{i}$, for $Cons(n)$, to i_n. We then run treatment learning to isolate a set of system level parameters I_n that have the highest likelihood of affecting unit level parameters i_n and determine whether or not the data points in V and v that lead us through the node n have a smooth relationship for function fitting.
2. *Line 6* If the relationship is smooth we build the map f_n such that $I_n = f_n(i_n)$.

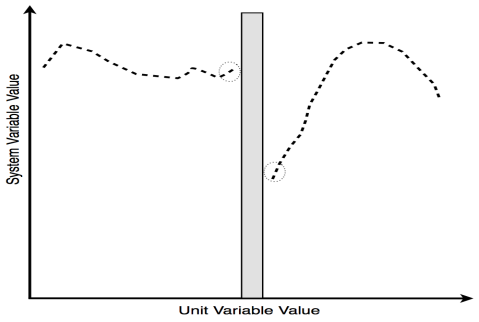

Figure 5: *A plot showing a non-smooth relationship between a system level and a unit level parameter. The gray region represents values of the unit level parameter which were not measured during experimental runs. The dotted circles surround the measured data closest in Euclidean distance to the desired unit level data.*

3. *Lines 8–10* If the relationship is not smooth, then we first strengthen constraint C by including the parent term from $Cons(n)$ and then recursively call $ComputerMap$ on the new constraint.
4. *Lines 12–22* If we cannot find a smooth relationship by including all terms in $Cons(n)$, then we use the Sufficiency of Witnesses to walk up the parent hierarchy of n to reach a node n'' that has at least Threshold data points that witness covering n''. We then attempt a linear fit using these data points and return.

We use the treatment learning algorithm TAR3, presented in Algorithm 3 for the following two purposes in our coverage algorithm.

Learning rules for covered nodes. We use TAR3 to tell us what subset of system level inputs and ranges covered nodes at the unit level. For every node n that was covered during system level testing, if its sibling node was also covered, then we have a partition of the data points at the system level into one set that covered n and the other set that covered the sibling of n. We use treatment learning with these partitions to learn rules that will either visit n or its sibling. These rules are learnt at Line 10 of Algorithm 1 and used subsequently at Line 16 to pick values for a subset of system level parameters as described in the algorithm.

Learning inputs for function fitting. When we attempt to fit a function to cover a node n, we start out with a weak constraint C that is initially set to $Term(n)$. This constraint is progressively strengthened as we see in Algorithm 2. For each constraint, we construct contrasting sets by partitioning the data points into a set with 20% of the data points that are closest in Euclidean distance to the constraint space and all remaining points. We use these contrasting sets to learn a rule that gives us a small set of system level parameters that most influence data points closest to the constraint space. This reduces the number of parameters for function fitting.

As a simple example, examine Figure 5, in which the desired unit level parameter values are represented by the gray rectangle in the center of the plot. The dashes represent data pairs seen during program execution, and the dotted circles surround the data nearest the PC boundary. The data inside the circles will be identified to TAR3 as desirable data. TAR3 then returns the system level parameters that most affect the unit level data near the PC boundary; we use this subset of system level parameters for function fitting.

This step of treatment learning is also used to determine whether or not there exists a smooth relationship between a given set of system level and unit level parameters. Figure 5 illustrates a case in which the relationship between the unit level and system level pa-

Algorithm 3: $RunTAR3(I, V, V', V'')$

input : System level parameters I, system level vectors V and contrasting sets $V' \subset V$ and $V'' = V \setminus V'$.

output: $(I', R, smooth)$ where $I' \subseteq I$, R is a set of ranges for each parameter in I, $smooth$ is set to true by examining the dataset

1 Call TAR3 with V, V' and V'';
2 Compose $I' \subseteq I$, R and $smooth$ based on the results of running TAR3;
3 Return $(I', R, smooth)$;

rameters appears to be discontinuous. To the left of the PC boundary, we see that a relatively small variation in system level values leads to a relatively large variation in the unit level values, and that it is possible to get two different unit level values for the same system level value. To the right of the PC boundary, we have an identical situation though there is much wider variation in system level values. In Figure 1, we see the three-dimensional equivalent to Figure 5, where the data that leads to executions close to the PC boundary is boxed, and TAR3 has returned two system level parameters in its treatment. In this figure there are two clusters of boxed data, with a large number of undesired data points separating the clusters, suggesting the discontinuity. We handle this discontinuity by strengthening the constraint as described in Algorithm 2.

4.4 Discussion

We now discuss the assumptions that we make in our coverage algorithm and then discuss the conditions under which the algorithm makes progress. We make the following assumptions in our coverage algorithm:

1. The unit U can be analyzed using concolic execution,
2. There is at least one path that is taken in the unit during system level testing.

The first assumption is required since our approach is to use unit level concolic execution for system level testing. The second assumption may be satisfied using one of the following two approaches:

1. Iteratively choose smaller and smaller systems that enclose the unit, till we find a system that exercises at least one path in the unit during system level testing and
2. pick the earliest method U' up the call chain of the unit U that has at least one path covered during system testing and run $Cover(S, U')$. This increases the test vectors that explore unit U' and hence the likelihood of taking paths in U.

We remark that by using a breadth first exploration of the constraints tree, we ensure that when we attempt to cover a node, all its parent nodes have been processed using treatment learning. This ensures that when we build a system level vector for a node n, we have learnt ranges for all nodes in the parent hierarchy of n; the system level vector is composed using these ranges and the function we learnt to cover n.

REMARK 1. **(Progress)** *In the presence of perfect function fitting, if we have an over-approximation of the subset of system level parameters that affect the unit level parameters $i_n = Vars(Cons(n))$ for every node n that is satisfiable at the system level, then the algorithm will eventually cover n.*

Consider a node n, satisfiable at the system level, that cannot be covered by considering any constraint weaker than $Cons(n)$. Since we progressively strengthen the unit level constraint C from $Term(n)$ to $Cons(n)$, we eventually include in C all terms from $Cons(n)$ and all parameters, i_n, in $Vars(Cons(n))$. If now, we find a perfect function f, such that $I_n = f(i_n)$, then as long as I_n includes all the system level parameters that affect i_n, we are guaranteed to cover n. We use treatment learning to extract the subset of system level parameters I_n. This step can be supplanted with other static analysis techniques, such as [8], that learn an over-approximation of the set I_n. Note that due to loops or recursion, our algorithm may not terminate.

5. EXPERIENCE

In this section, we present our experience using the technique proposed in this paper on several examples. The function fitting attempts to learn a function f_n for each node n that we can cover at the unit level, such that $I_n \simeq f_n(i_n)$ for some subset of system level inputs I_n and unit level inputs i_n. In general, our fitted functions become more accurate as we collect more data, if the actual relationships between the desired i_n and I_n are locally smooth in the region defined by both the collected data values and the desired data values. This is borne out by our experience as we illustrate in the following case studies. We have implemented our algorithms in the context of analyzing C code. We note however, that for this work, we have visually inspected the graphical treatments to determine whether or not TAR3's treatments have contiguous clusters of data points that imply a smooth relationship between system and unit level inputs for function fitting. We propose to automate this process in future work.

5.1 A Piecewise Linear Case Study

As our baseline example we use the simple, piecewise linear implementation shown in Program 2. The function *Unit* is instrumented to perform concolic execution and graphical results are shown in Figure 6. The relationships between the system level and unit level parameters for this academic example can be determined explicitly, either by hand or automatically using a tool like concolic execution. All invocations of the Unit function begin at Node 1 in Figure 6. The relationship between the unit and system level parameters that determines whether control flow will pass from Node 1 to Node 2 or to Node 5 is $i1 = I2$. Therefore, if $I2 > 0$, then control flow will pass to Node 2 and if $I2 \leq 0$ then control flow will pass to Node 5. For our demonstration purposes, we treat the relationship between $i1$ and $I2$ as unknown, and try to determine it using heuristic methods.

Our initial trial is conducted by creating the 25 test cases composed of all combinations of $I1$ and $I2$ for the integer values bounded by -2 and 2 (Algorithm 1, Lines 2–4). After running these tests, we see that Node 4 and Node 7 within the unit are not covered, but we have the unit level constraints to cover them. The constraints tree is shown in Figure 4. Lines 2–4 show the unit-level parameters that the tree depends on: g2, g1, and i1. These correspond to the identically-named variables shown in Program 2. Lines 7–12 show the actual constraints tree for the unit. Line 12 corresponds to Node 7 in the constraint graph. The 'S' on that line refers to the constraint being satisfiable at the unit level. The first set of parentheses on line 12 give the unit level term necessary to cover the node.

We begin by doing a breadth-first search of the tree (Algorithm 1, Lines 7–14). Lines 7 and 10 in Figure 4 are covered sibling nodes (Algorithm 1, Line 8) and TAR3 returns the contrast rule set that will increase the likelihood that we pass through Node 2. TAR3 automatically uncovers the system level constraint $0.5 \leq I2 \leq 2$, as shown by the bars in Figure 7 outlining the highest-scoring treatment. Since TAR3 can only learn a rule for data that is seen, there will always be a lower and upper bound on every constraint. The boxed data in the plot is the 'desired' set in the

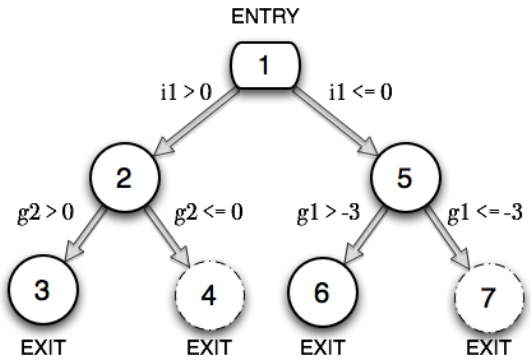

Figure 6: *The complete control flow tree for the prototype example Unit function given in Figure 4 after the initial testing. Covered nodes are represented by solid circles and nodes not covered have dotted outlines.*

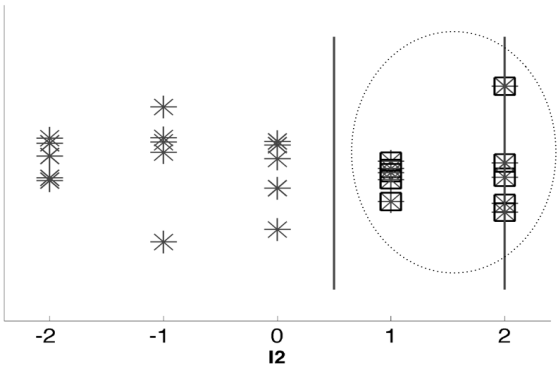

Figure 7: *The bars show the system-level constraints returned by TAR3 that we should use to guide execution through Node 2 within the unit. All data points are shown with an asterisk. The data that passed through Node 2 is boxed. In this case, it is possible to find a contiguous region for all of the boxed data, as shown by the dotted oval. If this had been a treatment for a node not covered in the constraints tree, a region like this would suggest that the relationship between the unit level input i and the system level inputs I returned by the treatment was smooth.*

two contrasting sets, and is the data that passed through Node 2. Similarly, TAR3 discovered the relationship ($-2 \leq I2 \leq 0.5$) necessary to guide execution through Node 5. Note that we do not expect TAR3 to exactly capture the location of the discontinuity; the system constraint boundary between Node 2 and Node 5 will depend on TAR3's discretization locations. Moreover, while Node 3 and Node 6 are covered nodes, we cannot use TAR3 to learn the necessary conditions for passing through them, since there is no contrasting data to learn from; Node 3 is covered, but its sibling Node 4 is not covered and similarly Node 6 is covered but Node 7 is not covered.

In order to build a relationship between nodes to be covered and system level inputs we begin by finding the Euclidean distance between each test case and the constraint boundary for each such node. We build two contrasting data sets by looking for the 20% of data that is closest to the constraint boundary. For Node 4, TAR3 suggests that the value of $g2$ depends only on $I1$, and that the relationship between the two is smooth. To cover Node 7, we begin with the weakest constraint, namely just the term $g1 \leq -3$. For Node 7, the top-scoring treatment results are shown in Figure 8. Here we see that the desired data set consisted of five data points spread discontinuously across the I1 and I2 space, and the top-scoring treatment made a prediction involving only three of the points. This is likely to happen when the entire data set is used and the relationship between the unit level parameters of interest and the system level parameters are not smooth; in this case, the functional relationship between $g1$ and $I2$ has a discontinuity at $I1 = 0$. We therefore should strengthen the constraint with the term $i1 \leq 0$ since we cannot find a smooth relation by considering just the term $g1 \leq -3$. By the Monotonicity of Witnesses, this yields fewer data points; there are a total of 15 data points that passed through Node 5. We now chose as one of our two contrast sets the 3 tests that come closest to the constraint boundary, namely $i1 \leq 0 \wedge g1 \leq -3$. TAR3's top-scoring result suggested that $g1$ depends on both $I1$ and $I2$ and that the relationship for this restricted set is now smooth.

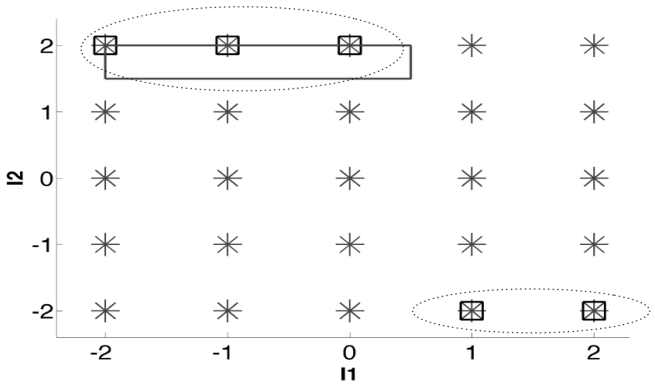

Figure 8: *The highest scoring treatment for Node 7. All data points are shown with an asterisk, and there are boxes around the data closest to the constraint boundary. The treatment suggests that the value of $g1$ depends upon both I1 and I2. It is not possible to find a contiguous region that contains all of the boxed data, suggesting that the relationship between the value of $g1$ and the system level parameters is not smooth.*

We now build approximations between the unit level parameters that occur in the constraint we consider for each node and system level parameters using Algorithm 2. We function fit between $g1$, $I1$, and $I2$. In this case, the linear least squares regression analysis has an error less than 10^{-15} and predicts $g1 = I2$, which for this simple test case is the exact solution. Cubic function fitting for the other node we want to cover, namely Node 4, gives us $g2 = I1 + 3$ with an error less than 10^{-14}.

For experimentation purposes, using all 15 of the test runs that passed through Node 5 results in an incorrect fit of $g1 = -0.6 * I1 - 0.2 * I2$; the discontinuity creates errors in the function fitting. Had we used this incorrect fit in order to generate new system level test cases we would have been unlikely to cover Node 7. By restricting the data to those that pass through Node 5 and come closest to satisfying the term $g1 \leq -3$ corresponding to Node 7, we learn an exact function in this case.

We now use our approximations between i and I to build new test cases (Algorithm 1, Lines 15–17). For this simple test case, the function fitting was exact and allowed us to quickly find the correct solutions for $I1$ and $I2$. We note that since we must pass through the ancestors before we can cover a node we combine the system level parameter ranges we found previously, (Algorithm 1,

Program 3 *The System Function in the Prototype Quadratic Example. The Unit Function is the same as in Program 2, except that the Unit Function for this case expects inputs of type* double.

```
double g1=1.0, g2=2.0;
int System(double I1, double I2)
{
  if (I1 > 0) g1 = I2; else g1 = -I2;
  g2 = I1*I2+3.0*I1*I1+I2*I2;
  Unit(I2, I1);
}
```

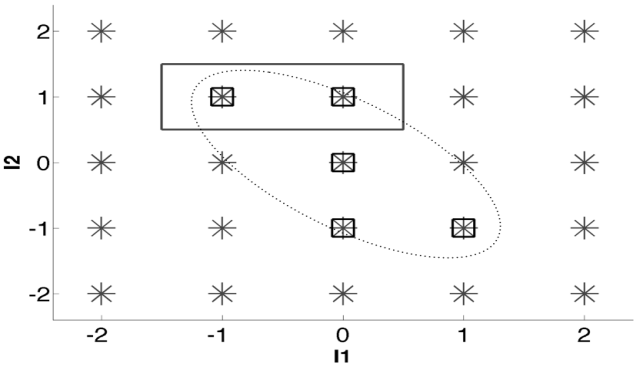

Figure 9: *The top-scoring treatment for Node 4. All data points are shown with a asterisk. The data closest to the constraint boundary is boxed. In this case, the boxed data is contiguous and the treatment implicates both system level parameters.*

Line 10), with values determined through function fitting to pass through both the ancestors and the node we want to cover.

5.2 A Piecewise Quadratic Case Study

The linear test case detailed in 5.1 has the advantage of being easily understood, but does not show how we can deal with complex, non-linear constraints, which are known to be problematic for concolic or symbolic execution approaches. Our technique is more likely to be of benefit when the relationships between the system and the monitored unit are complicated and/or nonlinear. As a simple example of how our technique could be used for nonlinear constraints, we propose the example shown in Program 3. The only differences between Program 3 and Program 2 are the use of *doubles* instead of *ints* as a variable type and the nonlinear formula for the assignment of $g2$ before *Unit* is called in the system. For this case study, the constraints tree is identical to the one given in Figure 4 and Figure 6. Our breadth-first search over the covered nodes gives us identical results to those found in 4.3.1.

We will build relationships for nodes N4 and N7. The results from TAR3 for Node 4 are shown in Figure 9. The boxed data is contiguous, suggesting that there may be a smooth relationship between the parameters. The fact that it is not possible to draw a box around only the desired contrast set suggests that the relationship between $g2$, $I1$, and $I2$ is likely to be nonlinear.

We function fit for the nodes Node 4 and Node 7. The case for Node 7 is identical to the piecewise linear case study. The interesting case is that of Node 4. For Node 4, the quadratic least squares fit gives a residual error of less than 10^{-15} for $g2 = 3.0 * I1^2 + I2^2 + I1 * I2$, which is the exact solution. The next step is to conjoin our constraints in order to find new test cases. When we attempt to create a system level vector that satisfies $g2 \leq 0$ and is consistent with the ranges for the system level parameters we learned in Line 10 of Algorithm 1, we discover that there is an inconsistency. There are no real roots that satisfy the constraint for $g2$ given the approximating function and the range constraints for the parent of Node 4. Using function fitting for the parent of Node 4, namely Node 2, we see that with a residual error on the order of 10^{-15} we get $i1 = I2$, the exact result. By simple substitution the correct system-level constraint is $I2 > 0$. A close examination of the constraint to satisfy Node 4 reveals that the two system level constraints are unsatisfiable; there is no system level test that will lead us to Node 4.

5.3 A Physical Case Study

We now examine a case study from the aerodynamics domain. Assume we have code for a new, supersonic aircraft that is designed to fly between 30,000 and 80,000 feet at Mach numbers between 0.8 and 3.0. The Mach number Ma is a ratio of the airspeed of the plane to the speed of sound, and is calculated by measuring the ratio of the measured air pressure P_t to the static air pressure P_s. Values of Ma below 1 mean that the plane is traveling subsonically, while values above 1 mean that the plane is traveling supersonically. This aircraft has a novel control system, and the code uses a function to predict the drag coefficient C_d so that it can update several coefficients in the yaw control law. For our system, we implemented the part of the code that takes in three arguments from sensors: P_t, P_s, and the altitude Alt. It uses this sensed data to calculate Ma, compressible and incompressible skin friction coefficients C_f and C_{fb}, and the corresponding terminal skin friction coefficients C_fTerm and $C_{fb}Term$. For subsonic compressible flow, Ma is given by the following equation [2], where we have simplified the expression using the constants for air:

$$Ma = \sqrt{5\left[\left(\frac{P_t}{P_s}\right)^{\frac{0.4}{1.4}} - 1\right]}. \quad (1)$$

For supersonic flow, Ma is found implicitly using the *Rayleigh Pitot tube formula*, shown here as Equation 2.

$$\frac{P_t}{P_s} = \left(\frac{5.76Ma^2}{5.6Ma^2 - 0.8}\right)^{3.5} \frac{2.8Ma^2 - 0.4}{2.4} \quad (2)$$

This equation would normally be implemented in code as a look-up table. For our system, we solve the equation using Newton's Method and have checked the results against the table printed in [2]. The skin friction coefficients C_f, C_{fb}, C_fTerm and $C_{fb}Term$ are complicated nonlinear functions of Ma and Alt, and are found using equations from the USAF Stability and Control DATCOM manual. [13]

The unit calculates C_d based on the skin friction and the base drag, again based on basic DATCOM equations. The relationships between C_d and the unit inputs are nonlinear, but the constraints that determine what that relationship is are linear and trivial to solve using concolic execution. We begin our testing of the system by looking at nominal ranges for the aircraft: Alt between 30 and 80 thousand feet, P_t between 0.0145 and 25, and P_s between 0.00971 and 3.5. Performing 2-factor combinatorial testing [17] with 5 bins for each of these parameters gives us 9 initial test cases. Two of these cases have $P_t < P_s$, a physical impossibility, so those two test cases are thrown out. The constraints tree for our 7 initial test cases are shown in Figure 10; these tests cover only two paths through the tree. There are constraints not covered for $C_f \leq C_fTerm$ and for Mach numbers in the subsonic and transonic regime where $Ma \leq 1.04$.

We do a breadth-first search of the constraints tree. For the nodes at line 22 and line 18 of Figure 10 that are not covered, TAR3 sug-

```
  [Parameters]
2 CfbTerm
3 Cf
4 Ma
5 CfTerm
6 Cfb

  [Tree]
9   (Cf > CfTerm) (C, ...)
10     (Ma >= (780000 / 1000000)) (C, ...)
11       (Ma > (1040000 / 1000000)) (C, ...)
12         (Ma >= (600000 / 1000000)) (C, ...)
13           (Cfb > CfbTerm) (C, ...)
14             (Ma >= 1) (C, ...)
15               (Ma <= (2000000 / 1000000)) (C, ...)
16               (Ma > (2000000 / 1000000)) (C, ...)
17             (Ma < 1) (S, ...)
18           (Cfb <= CfbTerm) (S, ...)
19         (Ma < (600000 / 1000000)) (S, ...)
20       (Ma <= (1040000 / 1000000)) (S, ...)
21     (Ma < (780000 / 1000000)) (S, ...)
22   (Cf <= CfTerm) (S, ...)
```

Figure 10: *The constraints tree after seven rounds of initial testing*

gests that there may be a smooth relationship between the associated unit level parameters and the system level parameters P_s and Alt. For the constraints on line 17 and lines 19-21 of Figure 10, TAR3 suggests that there may be a smooth relationship between Ma and the system level parameters P_t and P_s. We have only 7 initial test cases and since TAR3 suggests that the variation in the unit level parameters near constraint boundaries of nodes not covered can be primarily explained by two of the three system level parameters, the highest-order polynomials we will be able to fit using the initial data will be quadratic, as shown in Table 1. We perform function-fitting for the nodes not covered by system level testing, first attempting to use all 7 initial data points. The initial approximation between Ma and the system level parameters is $Ma = 5.7022 + 0.0035 * P_t^2 - 0.0092 * P_s * P_t + 0.7255 * P_s^2 - 0.0124 * P_t - 3.4665 * P_s$ and the residual this approximation gives for our 7 initial test cases is 0.0479. We repeat this process to find approximations between the unit level parameters C_f, C_{fb}, C_fTerm and $C_{fb}Term$, and the system level parameters P_s and Alt that were implicated by TAR3.

We now solve to find test cases for each node not covered in the constraints tree. For each of those nodes in Figure 10 we end up with undetermined systems (more unknowns than equations). In addition, this is a complicated set of equations involving two nonlinear equalities and an inequality, so we attempt to solve the set of equations graphically. We create a large number of test cases and evaluate them, the approximations are no greater than quadratic, so the solution of 235,000 tests (including graphing) takes 1.37 seconds in MATLAB on a laptop computer. We repeat this process for every node we would like to cover; when we are unable to find a solution using the quadratic approximations, we use the linear approximations. We arbitrarily chose potential test cases and added them to our suite. The approximated relationship between C_fTerm and the system level parameters suggests that we expand the range of P_t to a max of 50. In this first test case expansion we create 17 new tests based on the approximated relationships between the system and unit level parameters, execute these tests, and use concolic execution to record the paths taken through the unit. The resulting constraints tree has 5 covered paths with 21 covered nodes and 12 nodes not covered—only 5 of the nodes not covered are satisfiable. When the constraints tree after one iteration is compared against the one in Figure 10, we see that the constraints at lines 18, 20 and 22 are now covered. After two rounds of initial testing, our method used 24 tests to illuminate a constraints tree with 21 covered nodes and 12 nodes not covered. As a comparison against state-of-the-art black box testing, we also generated a suite with 25 n-factor combinatorial tests. We assume that the n-factor combinatorial testing will obtain better coverage than pure combinatorial tests [9, 10]. The constraints tree generated using the 25 n-factor combinatorial tests has 16 covered nodes and 10 nodes not covered.

6. RELATED WORK

The work related to automated testing is vast and we only highlight here the work that is most related to our approach.

There are many approaches that use symbolic or concolic execution [29, 15, 26, 6] for automated test case generation. However, all these approaches apply at the unit level and they do not consider integration with system level testing, as we do here.

Our work is related to other hybrid approaches such as [30, 16]. These works combine abstraction techniques and theorem proving for program analysis and testing but do not address the problem of constraining the system level inputs for a more focused unit analysis.

The work on carving differential unit tests from system tests [11] extracts the components that influence the execution of a unit and reassembles them so that the unit can be exercised as it was by the system test. Differential unit tests are used to detect differences between multiple unit implementations and they can not be used to guide the system level inputs to increase coverage .

We use machine learning techniques to determine constraints on system level inputs that lead to coverage of various regions in the code under analysis. For the case studies presented here we used a simple analysis to determine correlations between system level inputs in terms of range restriction on unit inputs. However other learning techniques, such as the Daikon invariant detector tool [12], can be used for the same purpose.

Finally, in previous work [22] we described a symbolic execution framework that used system level simulations to improve the precision of symbolic execution at the unit level. This was achieved in two ways: first, the framework allows symbolic execution to be started at any point in the program; thus, the concrete execution of the system can be effectively used to set up the environment for the symbolic execution of a unit in the system. However, that work could not be used for *guiding* the generation of new system level inputs to increase the coverage of the unit—which is our contribution here. Furthermore, we describe in [22] how to use the data collected during system level runs to mine constraints on the unit level inputs (using treatment learning or Daikon, for example). While this approach would allow more focused unit level testing, it suffers from the drawback that the mined constraints can be unrealistically restrictive, and thus prevent us to achieve coverage of corner cases in the unit.

7. CONCLUSION

We described a novel testing technique that combines the strengths of black-box system simulation with white-box unit symbolic execution to overcome their weaknesses. The technique uses machine learning, function fitting and constraint solving to iteratively guide the generation of system-level inputs to increase the testing coverage. We have applied the techniques for testing complex code from the aerospace domain. In the future, we plan to study alternative approaches to the machine learning technique described here (e.g. Daikon) and to perform a tighter integration of the black-box and white-box techniques. We also plan to perform a thorough evaluation of the technique to determine its utility in practice.

8. REFERENCES

[1] A. Acevedo, J. Arnold, and W. Othon. ANTARES: Spacecraft simulation for multiple user communities and facilities. In *AIAA*, 2007.

[2] J. D. Anderson. *Fundamentals of Aerodynamics*. Mc-Graw Hill, third edition, 2001.

[3] R. Bartle. *The elements of real analysis*. John Wiley & Sons, second edition, 1976.

[4] S. Bay and M. Pazzani. Detecting change in categorical data: Mining contrast sets. In *KDDM*, 1999.

[5] R. L. Burden and J. D. Faires. *Numerical analysis*. Brooks/Cole, seventh edition, 2001.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM CCS*, 2006.

[7] C. Cai, A. Fu, C. Cheng, and W. Kwong. Mining association rules with weighted items. In *IDEAS*, 1998.

[8] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.

[9] D. Cohen, S. Dalal, J. Parelius, and G. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[10] I. Dunietz, W. Ehrlich, B. Szablak, C. Mallows, and A. Iannino. Applying design of experiments to software testing: experience report. In *ICSE*, pages 205–215, 1997.

[11] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE*, 2006.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3), 2007.

[13] R. Finck. USAF stability and control DATCOM. Technical Report AFWAL-TR-83-3048, USAF, 1978.

[14] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *ASE*, 2010.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*. ACM, 2005.

[16] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *FSE*, 2006.

[17] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In *AIAA Aerospace*, 2009.

[18] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11, 1993.

[19] J. C. King. Symbolic execution and program testing. *CACM*, 1976.

[20] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 1997.

[21] T. Menzies and Y. Hu. Data mining for very busy people. *IEEE Computer*, November 2003.

[22] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[23] R.Agrawal, T.Imeilinski, and A.Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD*, 1993.

[24] L. Schumaker. *Spline functions: basic theory*. Wiley-Interscience, 1981.

[25] K. Sen. Concolic testing. In *ASE*, 2007.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.

[27] G. Strang. *Linear algebra and its applications*. Thomson Learning, third edition, 1988.

[28] L. N. Trefethen and I. David Bau. *Numerical linear algebra*. SIAM, 1997.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, 2005.

[30] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, 2006.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
04-01-2011	Technical Memorandum	5/2009-3/2011

4. TITLE AND SUBTITLE

Symbolic Execution Enhanced System Testing

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Misty Davies, Corina Pasareanu, and Vishwanath Raman

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

534723.02.02.01.40

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Ames Research Center
Moffet Field, California 94035-1000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITOR'S ACRONYM(S)

NASA

11. SPONSORING/MONITORING REPORT NUMBER

NASA/TM-2011-215962

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 61
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We describe a novel testing technique that uses the information computed by a symbolic execution of a program unit to guide the generation of inputs to the system containing the unit, in such a way that the unit's, and hence the system's, coverage is increased. The symbolic execution is performed at run-time, along program paths obtained by system level simulations. We demonstrate the effectiveness of our technique on two illustrative examples and on a realistic example from the NASA domain.

15. SUBJECT TERMS

system testing, symbolic methods, simulation, machine learning, function fitting

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19b. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	16	**19b. TELEPHONE NUMBER *(Include area code)*** (443) 757-5802