Symbolic Execution Enhanced System Testing

Misty Davies[1], Corina S. Păsăreanu[2], and Vishwanath Raman[2]

[1] NASA Ames Research Center, Moffett Field CA 94035, USA
misty.d.davies@nasa.gov
[2] Carnegie Mellon University, Moffett Field, CA 94035, USA
corina.s.pasareanu@nasa.gov, vishwa.raman@west.cmu.edu

Abstract. We describe a testing technique that uses information computed by 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 computes unit constraints at run-time, along program paths obtained by system simulations. We use machine learning techniques –treatment learning and function fitting– to approximate the system input constraints that will lead to the satisfaction of the unit constraints. Execution of system input predictions either uncovers new code regions in the unit under analysis or provides information that can be used to improve the approximation. We have implemented the technique and we have demonstrated its effectiveness on several examples, including one from the aerospace domain.

1 Introduction

Modern software, and in particular flight control software like that written at NASA, needs to be highly reliable and hence thoroughly tested. NASA software is typically tested using system level Monte Carlo or combinatorial simulations. Such system level "black-box" simulations have the advantage that they are (a) easy to set up, since the user only needs to specify the ranges for the system level inputs, and (b) can be used to test software systems that contain COTS ("Commercial-Off-The-Shelf"), binary or even hardware components that are impervious to "white-box" methods. However, system level simulations provide few guarantees in terms of testing coverage. Furthermore, they may be quite expensive. For example, a run using NASA's ANTARES simulator [1] may take hours to complete.

Recently, a new set of techniques [2,3,4] based on symbolic execution [5] 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 (up to some user specified bound). Such white-box techniques are not effective in the presence of COTS or binary components; e.g., in such cases, concolic execution may lead to divergence [4]. For this reason, and due to the large number of paths to explore and complex constraints to be solved, white-box symbolic execution is used most effectively for testing individual software units, but not the whole system. On the other hand, when analyzing a unit in

R. Joshi, P. Müller, and A. Podelski (Eds.): VSSTE 2012, LNCS 7152, pp. 294–309, 2012.

isolation, it is often the case that the unit's inputs need to be constrained by the system calling context, in order to obtain realistic test cases. Encoding input constraints requires significant manual effort by developers [2].

The goal of our work is to find system level test cases that increase the coverage of a unit of interest by exploiting a synergy between black-box system simulation and white-box unit symbolic execution. 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 approach improves on system level testing by increasing the obtained coverage with a reduced number of tests, and hence with a reduced cost. It also enables a modular unit level analysis under *realistic* contexts, since symbolic execution is performed along the program paths obtained via simulation.

Specifically, we use data mining techniques (i.e. treatment learning [6]) to obtain an 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, together with the approximations, 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 in the context of the analysis of C programs. We report here on the application of our approach to several illustrative examples, including one from the aerospace domain.

Related Work. The work related to automated testing is vast and we only highlight here the work that is most related to our approach. We have already discussed related symbolic and concolic execution approaches [7,4,3,8]. The work on carving differential unit tests from system tests [9] 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; they can not be used to guide the system level inputs to increase coverage.

In previous work [2] 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 described in [2] 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.

2 Background

A Program Model. A program is a 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*, defined to be a type from the set $\{int, short, unsigned\ int, char, float, double, enum\}$, with each element $a \in I$ taking values from a domain D_a based on its type; all assignment and conditional statements refer to elements in I. The set of all executions of the program P is $R(P) \subseteq \{(A \cup C)^*\}$ – 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, called a *valuation*, is denoted by $\boldsymbol{I}$ and associates every element $a \in I$ to a value in D_a. Given a valuation $\boldsymbol{I}$, we assume that all executions of the program visit exactly the same finite sequence of assignments and conditional statements; the programs are deterministic.

Concolic Execution. Concolic execution [4,10] is a technique that combines concrete and symbolic program execution to increase path coverage. Symbolic path constraints (PCs) are collected along concrete program runs; the PCs are conjunctions of Boolean expressions, each expression representing the condition on the inputs to follow that particular path. The conditions in the PCs are systematically negated to generate new PCs that are solved with off-the-shelf solvers. The obtained solutions are used as new program inputs to run the program along different paths. The process terminates when all the paths have been resolved or a user-specified bound has been reached; paths are either covered, unsatisfiable or unsolvable due to limitations in the chosen solvers.

Treatment Learning. Treatment learning [6,11] is a machine learning technique that finds the minimal difference between two sets. In our work, we use treatment learning to determine a *small number* of controllable inputs and ranges (a *treatment*) that are most likely to lead to some output.

TAR3 is a treatment learner that finds association rules involving both continuous and discrete variables quickly [11]. 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 (subsets of input parameters and their ranges) that maximize the likelihood of seeing points in the desired set. We note that one can use other association rule learners [12,13,14] to potentially find more accurate rules; however this would come with greater complexity and time costs [15,16].

Function Fitting. Function fitting finds a predictive relationship between associated outputs and inputs (usually one output variable and a small number of inputs). We use discrete least-squares function fitting [17,18] to approximate a relationship between the unit inputs and the associated system inputs; the technique is less sensitive to outliers than many competing techniques [19]. Assume $y(x)$ is a complex, non-linear function; its approximation can be given by a polynomial $p(x)$ with coefficients c_i, for $i \in \{1, 2, 3, \ldots\}$. A least-squares solution finds the constant values c_i that minimize the total Euclidean distance (the *residual*) between $p(x)$ and $y(x)$ at the given measurements x. If the relationships we are trying to approximate are Lipschitz continuous (or *smooth*), we can find a polynomial approximation that is arbitrarily close to our desired function by the Weierstrass Approximation Theorem [20]. A function that is not smooth

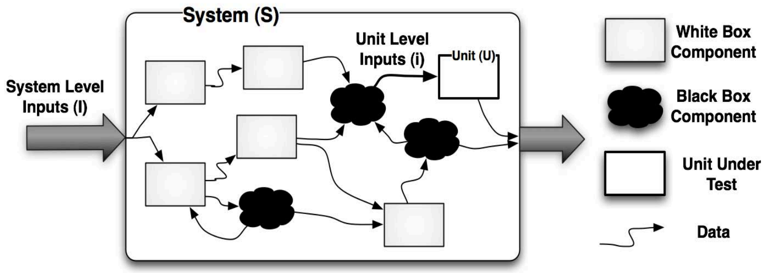

Fig. 1. A system S with inputs I and an embedded unit U with inputs i

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*. Shrinking each subinterval allows for arbitrarily close approximations with low-order polynomials [21]. We use the term *Threshold* to represent the minimum number of data points that we need for function fitting.

3 Approach

We illustrate the proposed approach using Figure 1, which shows a System Under Test (S) that may have both white-box and black-box components. A white-box unit is a code fragment that lends itself to concolic testing. S is a system with input parameters I containing a white-box unit $U = (i, A, C)$ with unit level parameters i. The goal is to generate system level inputs I that increase the coverage of 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; this constraint is obtained by the concolic execution of U. As an example, if $i = \{v, w\}$, a constraint could be $(v > w)$. We note that the concolic execution of U (in isolation) excludes the system that instantiates U; while this is useful for discovering new constraints for the uncovered paths, it may also generate an over-approximation of the actual 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 S. If $Cons(c)$ is satisfiable, then a satisfying valuation $\boldsymbol{i}$ will enable us to cover statement c at the unit level, but as mentioned, that statement may still be unreachable at the system level. Our goal is to try to generate assignments over the system level parameters I that can cover c (and other statements in the unit) during *system level testing*.

We note that the calling context for the unit can be represented by some function f such that $i = f(I)$. To discover the new valuations for I, we monitor the values of I and i during simulations and use machine learning techniques to approximate f, based on the monitored values. Once we have an approximation p of f, we use it to solve $i = p(I) \wedge Cons(c)$; the solutions for I are the likely candidates to the system level inputs that lead to the satisfaction of $Cons(c)$. These valuations are used to start new simulation runs, which lead to either covering c or to obtaining a more accurate

Program 1. 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;
  }
}
```

approximation of f. The process is repeated until either the desired coverage is obtained or a user-specified bound has been reached. We note here that if the function relating I and i is invertible, one can learn an approximation of the form $I = p(i)$ and use the solutions of $Cons(c)$ to directly obtain the valuations of I. To simplify the presentation, we will assume for the rest of the paper that we have such invertible functions. We describe our approach in detail in the next section.

4 Testing Algorithms

As a running example, consider the linear code in listing Program 1. Integers $I1$ and $I2$ are the system inputs, while $i1$ and $i2$ are the unit inputs. The two integer global variables $g1$ and $g2$ are treated as inputs to both System and Unit. The unit inputs are therefore $i1$, $i2$, $g1$ and $g2$.

Constraints Trees. We assume concolic execution achieves full path coverage over Unit. The set of path constraints over all executions of Unit are stored in a *constraints tree* T. The constraints tree reflects the set of all paths that were taken by all executions of a program unit (assume that the unit has no infinite loops).

```
[Parameters]
i1
g2
g1
[Tree]
  (i1 > 0) (C)
    (g2 > 0) (C)
    (g2 <= 0) (S)
  (i1 <= 0) (C)
    ((g1 + 3) > 0) (C)
    ((g1 + 3) <= 0) (S)
```

Fig. 2. The constraints tree after some rounds of initial testing

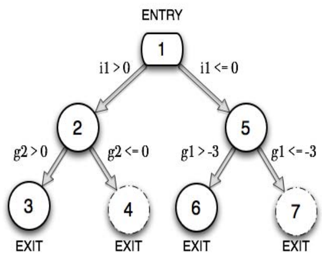

Fig. 3. A graphical representation of Figure 2. Covered nodes are solid circles; those not covered are dotted.

Figure 2 shows T for Unit after some initial testing. Lines 2-4 list the inputs that are constrained. Lines 6-11 contain a textual representation of the tree. The number of leaves is equal to the number of path constraints in T; each path constraint is a conjunction of the terms encountered along the parent hierarchy starting at each leaf. Therefore, given the tree in Fig 2, 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, denoted by the letter "C" within parentheses. The other constraints are satisfiable at the Unit level but not covered during system level testing, denoted by the letter "S".

Observations. Consider again a system S, with system inputs I, and a unit U within S, with unit inputs i. We let $d = |I|$. We assume the unit can be fully 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. We attempt to cover such nodes using a combination of concolic execution, treatment learning and function fitting. For a node n in T we take $Cons(n)$ as the unit constraint that leads to n and that when satisfiable will cover n. To present our coverage algorithm, we first make the following observations.

Consider a path $\sigma = n_1, n_2, \ldots, n_k$ in T such that all nodes n_i for $1 \leq i \leq k$ are covered by system testing. There exist vectors at the system and unit level that witness covering each node n_i in σ; for a set of system vectors V_i that witness covering n_i in σ, there exist corresponding witnesses v_i of unit vectors. 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 V_j and v_j is smooth for function fitting, then for all $i \geq j$, the relation between V_i and v_i is also smooth for function fitting.

Consider T and a $\sigma = n_1, n_2, \ldots, n_k$ in T such that all nodes that precede n_k are covered during system testing, but node n_k is not covered. Since concolic execution fails at the system level, we have that $Cons(n_k)$ is the finest symbolic path constraint, such that when $Cons(n_k)$ is satisfiable, the assignment that satisfies $Cons(n_k)$ covers n_k at the unit level. We take $Term(n_k)$ as the term corresponding to n_k and $Parent(n_k)$ as the parent of n_k in σ. Given a constraint C, let $Vars(C)$ be the set of parameters that appear in the terms of constraint C. The path constraint $Cons(n_k)$ is then $Term(n_1) \wedge Term(n_2) \wedge \ldots \wedge Term(n_k)$. We would like to learn the system level behavior as a function f, such that $I = f(Vars(Cons(n_k))$, via function fitting. If $Cons(n_k)$

is satisfiable, we can use f to find a system level vector that covers n_k using the satisfying assignment over $Vars(Cons(n_k))$ for $Cons(n_k)$. The caveat in this approach is that function fitting is 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 function fit for C, starting at $Term(n_k)$, progressively conjoining terms $Term(n_i)$ for $i = k-1, k-2, \ldots, 1$, stopping when we find a smooth function. This reduces the number of unit vectors we consider and by the Sufficiency of Witnesses considers the smaller number of data points.
– We reduce the number of system parameters for function fitting using treatment learning. For C, we use the data seen during system testing to find the subset $I_n \subseteq I$ of system parameters that most affect the values of the unit parameters in $Vars(C)$.

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 satisfying assignments. By the Monotonicity of Witnesses, we have more data points to cover these terms than to cover $Cons(n_k)$, increasing the likelihood of finding good treatments.

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 the d input parameters constrained by their data types. Unlike 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 [22]. For every system vector a, we monitor the unit and capture the unit vector b together with the path constraint for the path taken within the unit. The set of path constraints are summarized in T; system and unit vectors are stored in sets V and v.
2. *Lines 7–11*. We traverse the nodes in T in breadth first order. The treatment learner learns a treatment for each node n in T as long as its sibling is also covered. Since the treatment learner is a contrast set learner, it can be used to identify a set $I_n \subseteq I$ and ranges R_n of parameters in I_n, only when given data points that differentiate n from its sibling.
3. *Lines 13–16*. For each satisfiable node n in T not covered by MC simulations, we store the assignment $\boldsymbol{i}$ satisfying $Cons(n)$. We start with a constraint C set to $Term(n)$ and progressively strengthen C until we find a system vector to cover n. As we want to fit a function that maps I to i, we keep track of the parameters in C in i_n and the restriction of $\boldsymbol{i}$ to the parameters i_n in $\boldsymbol{i_n}$. The function $ComputeMap$ finds a function f_n such that $I_n = f_n(i_n)$ using function fitting.
4. *Lines 17–19*. We iterate over all satisfiable nodes n in T not covered during system testing. For each such n we run a system level test by composing a system vector as follows: (a) take $\boldsymbol{I_n} = f_n(\boldsymbol{i_n})$ 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.

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 in T using BFS) do
8          if (n and n's sibling are covered) then
               // Use contrasting data to 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             ∀j ∈ I_n store the range r_j ∈ R_n for j;
12         else
13             if (n is satisfiable but not covered) then
                   // Compute f_n such that I_n = f_n(i_n)
14                 i ← model for Cons(n);
15                 C ← Term(n);
16                 (I_n, i_n, f_n) ← ComputeMap(C, I, V, v, n, Parent(n), i);
       // Build new test-cases
17     for (n in T satisfiable but not covered) do
18         Run S with a consistent valuation using f_n(i_n) and ∀j ∈ I \ I_n using r_j
           from Line 10;
19         T' ← T' ∪ (PC from U);
20     T ← T';
21 until (T has no unprocessed nodes);
```

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

1. *Lines 1–4* We compute i_n occurring in C and the restriction of the model $\boldsymbol{i}$, for $Cons(n)$, to i_n. We use treatment learning to isolate a set $I_n \subseteq I$ most likely to affect i_n and to determine if the data points in V and v have a smooth relationship.
2. *Lines 5–6* If the relationship is smooth we build the map f_n such that $I_n = f_n(i_n)$.
3. *Lines 8–10* If the relationship is not smooth, we strengthen C by including the parent term from $Cons(n)$ and then recursively call $ComputeMap$.
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''. By Assumption 2, we have at least one path that was taken through the unit during system testing. If we find two data points that covered a node in the parent hierarchy of n, we attempt a linear fit and return. If we cannot find at least two data points, we run more MC simulations.

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

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

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

$i_n \leftarrow Vars(C)$;
$\boldsymbol{i_n} \leftarrow$ restriction of $\boldsymbol{i}$ to i_n;
`// Find a subset of` I `for function fitting`
$V' \leftarrow \{a \in V \mid a$ is in 20% of points closest to $Cons(n)\}$ and $V'' \leftarrow V \setminus V'$;
$(I_n, R_n, smooth) \leftarrow RunTAR3(I, V, V', V'')$;
if *(smooth)* **then**
    Build map $I_n = f_n(i_n)$;
else
    `// Strengthen constraint and try again`
    if *(n' exists)* **then**
        $C \leftarrow C \wedge Term(n')$;
        $(I_n, i_n, f_n) \leftarrow ComputeMap(C, I, V, v, n, parent(n'), \boldsymbol{i})$;
    else
        `// If no smooth relation between` I_n `and` i_n`, then walk up the parent of` n`, pick a node with Threshold points, and attempt a linear fit`
        $n'' \leftarrow n$;
        while *($Parent(n'')$ exists)* **do**
            $C \leftarrow C \wedge Term(Parent(n''))$;
            $n'' \leftarrow Parent(n'')$;
            $V' \leftarrow \{a \in V \mid a$ covers $n''\}$;
            if *($|V'| \geq Threshold$)* **then**
                break;
        $V'' \leftarrow V \setminus V'$;
        $(I_n, R_n, \_) \leftarrow RunTAR3(I, V, V', V'')$;
        $i_n \leftarrow Vars(C)$;
        Build map $I_n = f_n(i_n)$;

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 determine the subset of system inputs and their ranges that covered nodes at the unit level. For every node n that was covered during system testing, if its sibling 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 its sibling. We use TAR3 with these partitions to learn rules that will either visit n or its sibling; Line 10 of Algorithm 1. We use these rules at Line 18 to pick values for a subset of I as described in the algorithm.

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

input : System level parameters I, system level vectors V and contrast 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 output

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

Learning Inputs for Function Fitting. We attempt to fit a function to cover node n using a weak C initially set to $Term(n)$. This C is progressively strengthened as seen in Algorithm 2. For each C, we construct contrast sets by partitioning the data points into a.) the 20% of the data points nearest in Euclidean distance to the PC boundary and b.) all remaining points. These sets are used to learn a small subset of I most influencing i close to the PC boundary. We use this reduced subset of I for function fitting.

As an example, in Figure 4 the desired $\boldsymbol{i}$ are represented by the gray rectangle in the center of the plot. Curves are built from data pairs seen during program execution; dotted circles surround the data nearest the PC boundary and comprise a contrast set. TAR3 returns the I that most affect the i near the PC boundary. We also use TAR3 to determine whether a smooth relationship exists between subsets of i and I. In Figure 4, the relationship between i and I appears to be discontinuous. To each side of the PC boundary, a small variation in system values leads to a large variation in the unit values; it is possible to get two different unit values for the same system level value.

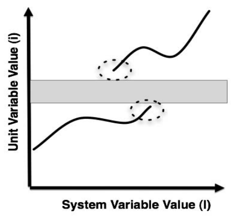

Fig. 4. A non-smooth relationship between a system and a unit parameter. The gray region represents values of $\boldsymbol{i}$ not seen during testing. Dotted circles surround data closest to the boundary.

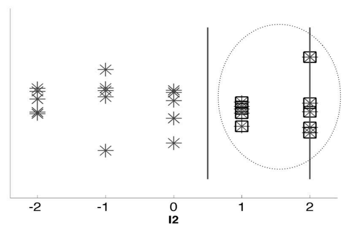

Fig. 5. Bars outline a rule that guides execution through Node 2. Data points (asterisks) are boxed if the runs pass through Node 2. The dotted oval outlines a contiguous region that suggests f_2 is smooth.

Discussion. We now discuss the assumptions made in our coverage algorithm and also 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. At least one path in U is taken during system testing.

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

1. Iteratively choose smaller systems that enclose U, until we find a system such that at least one path is taken in U during system testing.
2. Pick the earliest method U' up the call chain of U that has at least one path covered during system testing and then run $Cover(S, U')$. This increases the test vectors that explore 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. This ensures that when we build a system level vector for a node n, we have learnt ranges for all nodes in its parent hierarchy; the system level vector is composed using these ranges and the function f_n.

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

Consider a satisfiable node n that cannot be covered by considering any constraint weaker than $Cons(n)$. As we strengthen the C from $Term(n)$ to $Cons(n)$, we eventually include in C all terms from $Cons(n)$ and all i_n in $Vars(Cons(n))$. If we find a perfect function f, such that $I_n = f(i_n)$, and if I_n includes all the I that affect i_n, we are guaranteed to cover n. We use TAR3 to extract I_n. We can supplant TAR3 with static analysis techniques, such as [23], to 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. Two of these examples are purely illustrative, the third is a classic aerospace example. Planned experiments include larger aerospace examples: flight control software for unmanned aerial vehicles and a prototype conflict detection and resolution algorithm.

Our algorithms are implemented in the context of analyzing C code. We use MATLAB scripts to generate an initial suite of system vectors V given the known I, and to execute programs instrumented for concolic execution. The concolic execution framework is implemented using CIL [24], the C Intermediate Language, that provides an API for the analysis of C programs, to instrument user code. We use CIL to walk the intermediate representation of the program and insert calls to a set of runtime listeners. The user program is then re-generated from the intermediate representation, linked with our runtime library and run. During MC simulations, we use the instrumented version of the unit to monitor unit and system inputs and to capture paths that were taken within the unit. The constraints tree generated during MC simulations is used as an input to a subsequent solve cycle, where we solve for paths not taken within the unit during system level testing, replay solutions found and thus explore the tree to completion; we solve path constraints using Yices [25]. The outputs of these steps are a fully explored constraints tree T together with models for all satisfiable paths, a set of unit vectors v and the corresponding system vectors V that we monitored during MC simulations. These outputs are fed to MATLAB scripts that use I, T, i, V and v to perform treatment learning and function fitting, and to predict new $\boldsymbol{I}$ that better cover T in subsequent iterations. Two steps in our current process are manual, and we have plans to automate both: a) determining whether TAR3's treatments suggest smooth functions, and b) choosing whether to begin execution of the new $\boldsymbol{I}$.

A Piecewise Linear Case Study. We will first use the simple, piecewise linear implementation in Program 1. Although the f_n for this example can be found by hand or by symbolic execution, we use it here to illustrate our technique. *Unit* is instrumented to perform concolic execution and graphical results are shown in Figure 3. All invocations of Unit begin at Node 1 in Figure 3. Control flow from Node 1 is determined by f_1, which is $i1 = I2$. If $I2 > 0$, control flow passes to Node 2; otherwise, to Node 5. For demonstration, we treat f_1 as unknown, and determine it using our heuristic methods.

We initially create 25 test cases using values for $I1$ and $I2$ between -2 and 2 (Algorithm 1, Lines 2–4). Nodes 4 and 7 within Unit are not covered; concolic execution provides the unit input constraints that will cover them. Figure 2, Lines 2–4 give the required unit level parameters: g2, g1, and i1. Lines 6–11 show T for Unit; Line 11 corresponds to Node 7, and has an 'S' to show that the constraint is satisfiable at the unit level.

The generated constraint tree is traversed using breadth-first search (Algorithm 1, Lines 7–16). Lines 6 and 9 in Figure 2 indicate covered sibling nodes (Algorithm 1, Line 8); TAR3 automatically returns the rule set for passing through Node 2, $(0.5 \leq I2 \leq 2)$, as shown by parallel bars in Figure 5. Similarly, TAR3 discovers $(-2 \leq I2 \leq 0.5)$ for passing through Node 5. Note that TAR3 does not capture the exact location of the constraint boundary between Nodes 2 and 5. TAR3 can not learn system constraints for Nodes 3 and 6 as there is no contrasting data.

TAR3 is then used to reduce the subset of values of I_n for function fitting. Contrast data sets are built by isolating the 20% of unit input data nearest the constraint boundary. For Node 4, TAR3 suggests that $g2$ depends on a smooth relationship involving only $I1$. To cover Node 7, our approach first considers all data satisfying the weakest constraint $(g1 \leq -3)$; TAR3's results are in Figure 6. The data nearest in value to the constraint boundary are spread discontinuously across I1 and I2 space. TAR3 makes a prediction involving a subset of the points. This happens when the the relationship between i and I is not smooth; in this case, the relationship between $g1$ and $I2$ has a discontinuity at $I1 = 0$. The constraint is strengthened by considering the data satisfying $i1 \leq 0 \wedge g1 \leq -3$. By the Monotonicity of Witnesses, this yields fewer data points; there are a total of 15 data points passing through Node 5. TAR3 now suggests there is a smooth $g1 = f_7(I1, I2)$.

For Node 7 the exact solution $g1 = I2$ is predicted using function fitting (Algorithm 2), with an error less than 10^{-15}. For Node 4 the solution $g2 = I1 + 3$ is predicted with an error less than 10^{-14}. These approximations, along with the previously discovered system level constraints (Algorithm 1, Line 10), enable building new test inputs for $I1$ and $I2$ to cover Nodes 4 and 7 on the next test iteration (Algorithm 1, Lines 17–19).

Program 2. The System Function in the Prototype Quadratic Example. The Unit Function is the same as in Program 1, 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);
}
```

A Piecewise Quadratic Case Study. As a simple example of how our technique could be used in the presence of nonlinear constraints (that are not typically handled by off-the-shelf solvers), we propose the example in Program 2. Program 2 and Program 1 differ in the use of *doubles* instead of *ints* and the nonlinear assignment formula for $g2$ before *Unit* is called. T is identical to the one given in Figure 2 and Figure 3. A breadth-first search over covered nodes gives identical results to the previous section.

TAR3's results for Node 4 are shown in Figure 7. The treatment was unable to bound all of the contiguous boxed data; this suggests that f_4 is smooth but nonlinear.

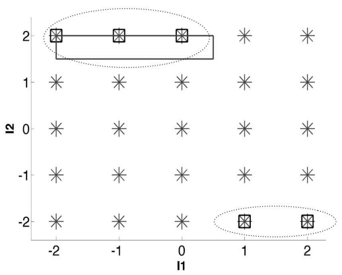

Fig. 6. Node 7's treatment

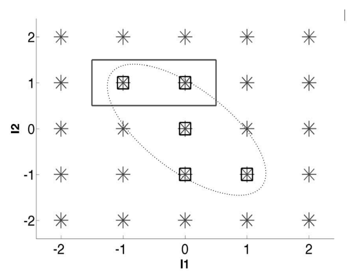

Fig. 7. Node 4's treatment

Function fitting is applied for Nodes 4 and 7. Node 7's results are identical to those in the previous section. For Node 4, function fitting gives a residual error of less than 10^{-15} and the exact solution $g2 = 3.0 * I1^2 + I2^2 + I1 * I2$. Our algorithm first attempts to create an $\boldsymbol{I}$ that satisfies $g2 \leq 0$ and is consistent with the system parameters and ranges learned previously (Line 10 of Algorithm 1), but discovers that there is an inconsistency. There are no real roots that satisfy the constraint for $g2$ given f_4 and the range constraints for Node 4's parent (Node 2). Function fitting for Node 2 yields the exact result $i1 = I2$. By simple substitution the correct system constraint is $I2 > 0$. An examination of Node 4's constraint reveals that the two system constraints are unsatisfiable; no system test leads us to Node 4.

An Aerodynamics Case Study. In this aerodynamics case study the code predicts the drag coefficient C_d, as calculated by the USAF Stability and Control DATCOM manual [26]; it can be found at `https://c3.nasa.gov/dashlink/projects/57/#c0`. C_d is used in the yaw control law for a supersonic aircraft designed to fly between 30,000 and 80,000 feet at Mach numbers M between 0.8 and 3.0. M is a ratio of the plane's airspeed to the speed of sound, and is calculated by measuring two different pressures, P_t and P_s. The system I consists of three arguments from sensors: P_t, P_s, and the altitude Alt. This sensed data is used to calculate M, compressible and incompressible skin friction coefficients C_f and C_{fb}, and the corresponding terminal skin friction coefficients C_fT and $C_{fb}T$. For subsonic ($M < 1$) compressible flow in air, M is given by Equation 1; for supersonic ($M >= 1$) flow, M is found implicitly using the *Rayleigh Pitot tube formula* [27], shown here as Equation 2.

$$M = \sqrt{5\left[\left(\frac{P_t}{P_s}\right)^{\frac{0.4}{1.4}} - 1\right]} \quad (1) \qquad \frac{P_t}{P_s} = \left(\frac{5.76M^2}{5.6M^2 - 0.8}\right)^{3.5} \frac{2.8M^2 - 0.4}{2.4} \quad (2)$$

For Equation 2, there is *no explicit formula* for M given P_t, P_s. One code component uses Newton's Method to solve Equation 2, and is used as a black box for our technique. C_f, C_{fb}, C_fT and $C_{fb}T$ are complicated nonlinear functions of M and Alt [26]. The unit calculates C_d based on the skin friction and the base drag. The relationships between C_d and the unit inputs are nonlinear, but the constraints defining the relationships are linear and easy to both discover and solve using concolic execution techniques.

```
  [Parameters]
2 CfbT
3 Cf
4 M
5 CfT
6 Cfb
  [Tree]
8  (Cf > CfT) (C)
9     (M >= (780000 / 1000000)) (C)
10      (M > (1040000 / 1000000)) (C)
11        (M >= (600000 / 1000000)) (C)
12          (Cfb > CfbT) (C)
13            (M >= 1) (C)
14              (M <= (2000000 / 1000000)) (C)
15              (M > (2000000 / 1000000)) (C)
16            (M < 1) (S)
17          (Cfb <= CfbT) (S)
18        (M < (600000 / 1000000)) (S)
19      (M <= (1040000 / 1000000)) (S)
20    (M < (780000 / 1000000)) (S)
21  (Cf <= CfT) (S)
```

Fig. 8. The constraints tree after seven rounds of initial testing

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 [28] with 5 bins for each of these parameters gives 9 initial test cases. Two of these cases have $P_t < P_s$, a physical impossibility, and are thrown out.

The constraints tree T for our 7 initial test cases covers only 2 paths through the tree, as shown in Figure 8. T is traversed using a breadth-first search. For the nodes at lines 21 and 17 of Figure 8, TAR3 suggests a smooth relationship between the unit parameters and the system parameters P_s and Alt. For the nodes at lines 16 and 18-20, TAR3 suggests a smooth relationship between M and the system parameters P_t and P_s. Function fitting is performed for the nodes not covered by system testing, using all 7 initial data points, giving the approximation $M = 5.7022 + 0.0035 * P_t^2 - 0.0092 * P_s * P_t + 0.7255 * P_s^2 - 0.0124 * P_t - 3.4665 * P_s$ with a residual of 0.0479. This process is repeated to find approximations between the unit parameters C_f, C_{fb}, C_fT and $C_{fb}T$, and the system parameters P_s and Alt that were implicated by TAR3.

Constraint solving is then used to find test inputs for each node not covered in T. The result is 17 new $\boldsymbol{I}$, which are used for new simulations. Concolic execution records the paths taken through the unit; the resulting T has 5 covered paths with 21 covered nodes and 12 nodes not covered—only 5 of the nodes not covered are satisfiable. When the new T is compared against the one in Figure 8; the constraints at lines 17, 19 and 21 are covered. After two rounds of testing, our method uses 24 tests to illuminate a constraints tree with 21 covered nodes and 12 nodes not covered.

We compared our technique against state-of-the-art black box testing by generating a test suite with 25 n-factor combinatorial tests; n-factor combinatorial testing typically obtains better coverage than random Monte Carlo testing [22,29]. With a comparable number of tests (24 vs. 25) our technique achieves significantly higher coverage (21 covered nodes) than the coverage obtained by n-factor combinatorial testing alone (16 covered nodes).

6 Conclusion

We described a 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 and increases the testing coverage. We showed in the experience section that we could use our tool to increase coverage of a unit using fewer test cases compared to state-of-the-art combinatorial testing. System level simulation can be expensive, and using information from white-box techniques allowed us to significantly decrease the time cost. White-box techniques, like concolic execution, may not scale to a full system. This is especially true when the system either contains non-linear components or contains components for which the source code is unavailable. Covering each white-box unit separately is an option, but there are likely to be test cases which are not possible given the constraints of the full system. As an example, the values of the Mach number and the friction coefficients in our aerodynamics case study are constrained by the measured values of the pressures P_t and P_s and the altitude. This means that, even though the Mach number and the friction coefficients are treated as independent inputs to our unit, the values of these variables cannot truly vary independently. If we performed only unit-level full coverage, we may miss dead code that is unreachable given the system, or we may spend too much time exploring behaviors in the unit that are not possible given the unit's true calling context. In the future, we plan to study alternative approaches to machine learning (e.g. Daikon) and to perform a thorough evaluation of the technique to determine its utility in practice.

References

1. Acevedo, A., Arnold, J., Othon, W., Berndt, J.: ANTARES: Spacecraft simulation for multiple user communities and facilities. In: AIAA 2007–6888 Mod. and Sim. (2007)
2. Pasareanu, C., Mehlitz, P., Bushnell, D., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA (2008)
3. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: FSE (2005)
4. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. ACM (2005)
5. King, J.C.: Symbolic execution and program testing. CACM (1976)
6. Menzies, T., Hu, Y.: Data mining for very busy people. IEEE Computer (2003)
7. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
8. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM CCS (2006)
9. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: FSE (2006)
10. Sen, K.: Concolic testing. In: ASE (2007)
11. Gay, G., Menzies, T., Davies, M., Gundy-Burlet, K.: Automatically finding the control variables for complex system behavior. In: ASE (2010)
12. Bay, S., Pazzani, M.: Detecting change in categorical data: Mining contrast sets. In: KDDM (1999)
13. Agrawal, R., Imeilinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD (1993)
14. Cai, C., Fu, A., Cheng, C., Kwong, W.: Mining association rules with weighted items. In: IDEAS (1998)
15. Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Machine Learning 11 (1993)
16. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artificial Intelligence (1997)
17. Trefethen, L.N., David Bau, I.: Numerical linear algebra. SIAM (1997)
18. Strang, G.: Linear algebra and its applications, 3rd edn. Thomson Learning (1988)
19. Burden, R.L., Faires, J.D.: Numerical analysis, 7th edn. Brooks/Cole (2001)
20. Bartle, R.: The elements of real analysis, 2nd edn. John Wiley & Sons (1976)
21. Schumaker, L.L.: Spline functions: basic theory. Wiley Interscience (1981)
22. Cohen, D., Dalal, S., Parelius, J., Patton, G.: The combinatorial design approach to automatic test generation. IEEE Software 13, 83–88 (1996)
23. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA (2007)
24. Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction, pp. 213–228 (2002)
25. Dutertre, B., Moura, L.D.: The YICES SMT solver. Technical report, SRI International (2006)
26. Finck, R.: USAF stability and control DATCOM. Technical Report AFWAL-TR-83-3048, USAF (1978)
27. Anderson, J.D.: Fundamentals of Aerodynamics, 3rd edn. Mc-Graw Hill (2001)
28. Gundy-Burlet, K., Schumann, J., Barrett, T., Menzies, T.: Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In: AIAA Aerospace (2009)
29. Dunietz, I., Ehrlich, W., Szablak, B., Mallows, C., Iannino, A.: Applying design of experiments to software testing: experience report. In: ICSE, pp. 205–215 (1997)