Abstraction-guided Test Generation: A Case Study

Thomas Ball
Testing, Verification and Measurement Research
Microsoft Research
http://www.research.microsoft.com/tvm/

November 25, 2003

Technical Report
MSR-TR-2003-86

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Abstraction-guided Test Generation: A Case Study

Thomas Ball
Testing, Verification and Measurement Research
Microsoft Research
http://www.research.microsoft.com/tvm/

November 25, 2003

Abstract

We define an automated *behavioral* approach to unit test generation for C code based on three steps: (1) predicate abstraction of the C code generates a boolean abstraction based on a set of observations (predicates); (2) reachability analysis of the boolean abstraction computes an overapproximation to the set of observable states and generates a small set of paths to cover these states; (3) SAT-based symbolic execution of the C code generates tests to cover the paths. Our approach generalizes a variety of test generation approaches based on covering code and deals naturally with the difficult issue of infeasible program paths that plagues many code-based test generation strategies. We explain our approach via a case study of generating test data for a small program and discuss its capabilities and limitations.

1 Introduction

Automatic test generation from code usually has as its goal to cover the statements or branches in a program. [8, 14] Our goal is to re-orient unit test generation to focus on covering important aspects of a program's *behavior* rather than its *structure*, and define an automated process to achieve high behavioral coverage. Good code coverage will be achieved as a side-effect of achieving good behavioral coverage. As we will show, our approach generalizes a variety of code coverage approaches and deals naturally with the difficult issue of infeasible program paths that plagues many code-based test generation strategies.

We use the reachable states of a program as a window into its behavior. A state is a mapping from locations to values (often associated with a program point). While a program may have an unbounded number of reachable states, at any point in time there usually are only a finite number of observations about these states that will be interesting to test. A *predicate* maps a state to a boolean value. For example, the predicate $(x > 0)$ observes whether or not variable x has a positive value in a given state. An observer o is a vector of n predicates. The observed state $o(s)$ corresponding to state s is a vector of Boolean values (paired with a program point) constructed by applying each predicate in o to state s. Predicates are a way to observe program behavior and bound the number of observed states, as there are a finite number of program

points in a program and a finite number of observations (2^n) that can be made at each program point.

We define an automatic process for creating test data to achieve high behavioral coverage of a sequential C program P. The input to the process is a set of predicates E observing states of P. (We will discuss later ways to select E). Our process creates concrete inputs to the program P that will cause it to reach a high percentage of the reachable observable states (induced by E). The process has three main steps (based on existing algorithms and tools):

- Using predicate abstraction [12], a boolean program $BP(P, E)$ is automatically created from a C program P and predicates E. [1] (A boolean program has all the control-flow constructs of C but only permits variables with boolean type.) Each predicate e_i in E has a corresponding boolean variable b_i in $BP(P, E)$ that conservatively tracks the value of e_i. The boolean program is an abstraction of program P in that every feasible execution path of P is a feasible execution path of $BP(P, E)$.

- Reachability analysis of the boolean program using the BEBOP symbolic model checker [2] yields an overapproximation S_E to the set of observable reachable states of P. We have modified BEBOP to output a small set of paths $P(S_E)$ that covers all states in S_E.

- A SAT-based symbolic simulator for C [7] determines if each path p_s in $P(S_E)$ is feasible in the source program P and generates an input to cover p_s if it is feasible.

All of the above three steps are completely automatic. The central problem is the selection of the predicates E. If E is the empty set then the boolean abstraction will be imprecise and very few paths chosen by the model checker will be feasible. Let $F(S_E)$ be the subset of paths in $P(S_E)$ that are feasible paths in the source program P. The ratio $|F(S_E)|/|P(S_E)|$ gives us a measure of the goodness of a set of observations E. The closer the ratio is to one, the closer the boolean abstraction $BP(P, E)$ approximates the program P. As we will see, choosing the set of predicates from the conditionals in a program is a good start to achieving a high coverage ratio.

One question that the reader may ask is "why bother generating inputs at all if the symbolic machinery is powerful enough to determine the feasibility of a program path?". There are several reasons. First, tests are a reusable asset understood by programmers and testers Programmers and testers will gain confidence in their program (and the symbolic machinery used to generate tests) by seeing their programs run and produce results predicted by the symbolic analysis. Second, a test generated by our analysis for a function f can be used to test alternate implementations of the function f. Finally, in the end, testing must be done to ensure proper end-to-end behavior of a system, despite the application of the best verification technology.

To bring these ideas into focus, we present a case study in which we apply the process to a small example. Section 2 presents the case study. Section 3 discusses the issues, problems and opportunities that emerge from this test generation process. Section 4 describes related work and Section 5 concludes the paper.

```
void partition(int a[], int n) {
    int pivot = a[0];
    int lo = 1;
    int hi = n-1;
    assume(n>2);
L0: while (lo <= hi) {
L1:   ;
L2:   while (a[lo] <= pivot) {
L3:     lo++;

L4:     ;
      }
L5:   while (a[hi] > pivot) {
L6:     hi--;

L7:     ;
      }
L8:   if (lo < hi) {
L9:     swap(a,lo,hi);
LA:     ;
      }
LB:   ;
    }
LC: ;
}
```

(a)

```
void partition() begin
    decl lt,le,al,ah;
    enforce (!(lt&!le)&
       !(!lt&le&((al&ah)|(!(al|ah)))));
    lt,le,al,ah := T,T,*,*;
L0: while (le) do
L1:   skip;
L2:   while (al) do
L3:     lt,le,al := ch(F,!lt),
                    ch(lt,!lt||!le), *;
L4:     skip;
      od
L5:   while (ah) do
L6:     lt,le,ah := ch(F,!lt),
                    ch(lt,!lt||!le), *;
L7:     skip;
      od
L8:   if (lt) then
L9:     al,ah := !ah,!al;
LA:     skip;
      fi
LB:   skip;
    od
LC: skip;
end
```

(b)

Figure 1: (a) The `partition` function and (b) its boolean program.

2 Case Study

Figure 1(a) presents a (buggy) example of QuickSort's `partition` function, a classic example that has been used to study test generation [5] We have added various control points and labels to the code for explanatory purposes. The goal of the function is to permute the elements of the input array so that the resulting array has two parts: the values in the first part are less than or equal to the chosen pivot value `a[0]`; the values in the second part are greater than the pivot value. There are two array bounds check missing in the code: the check at the **while** loop at label `L2` should be `(lo<=hi && a[lo]<=pivot)`; the check at the **while** loop at label `L5` should be `(lo<=hi && a[hi]>pivot)`.

2.1 Observations

There are thirteen labels in the `partition` function (`L0`-`LC`), but an unbounded number of paths. If we bound the number of iterations of each loop in the function to be no greater than k, than the total number of paths is $f(k) = 1 + (2(1+k)^2)^k$, which grows very quickly. Which of these many paths should be tested? Which are feasible and which are infeasible? Clearly, paths are not a very good way to approach testing of this function.

Instead of reasoning in terms of paths, we will use predicates to observe the states of

the `partition` function. Let us observe the four predicates that appear in the body of the function: `(lo<hi)`, `(lo<=hi)`, `(a[lo]<=pivot)`, and `(a[hi]>pivot)`. An observed state thus is a bit vector of length four (**lt**, **le**, **al**, **ah**), where **lt** corresponds to `(lo<hi)`, **le** corresponds to `(lo<=hi)`, **al** corresponds to `(a[lo]<=pivot)`, and **ah** corresponds to `(a[hi]>pivot)`. There only are ten feasible valuations for this vector, as six are infeasible because of correlations between the predicates:

- If `!(lo<hi)&&(lo<=hi)` then `(lo==hi)` and so exactly one of the predicates in the set { `(a[lo]<=pivot)`, `(a[hi]>pivot)` } must be true. Thus, the two valuations FTFF and FTTT are infeasible.
- Since `(lo<hi)` implies `(lo<=hi)`, the four valuations TFFF, TFTT, TFFT and TFTF are infeasible.

Since there are thirteen labels in the code and ten possible valuations, we have a state space of 130 observable states in the worst-case. However, as we will see in Section 2.3, the number of reachable observable states is far less.

2.2 Boolean Abstraction

Figure 1(b) shows the boolean program abstraction of the `partition` function with respect to the four observed predicates. This program can be automatically constructed using the C2BP tool [1] in the SLAM toolkit [3]. The boolean program has one variable (`lt`, `le`, `al`, `ah`) for each observed predicate. Statements in the boolean program conservatively update each boolean variable to track the value of its corresponding predicate. The `enforce` statement in the boolean program has the effect of putting an `assume` statement (with the same expression as the `enforce`) before and after each statement. The expression in the `enforce` statement rules out the six infeasible states listed in the previous section.

Boolean programs contain parallel assignment statements. The first such assignment in the boolean program captures the effect of the statements before label `L0` in the `partition` function:

```
lt,le,al,ah := T,T,*,*;
```

This assignment statement sets the values of variables `lt` and `le` to true because the C code before label `L0` establishes the conditions `n>2`, `lo==1`, and `hi==n-1`, which implies that `lo<hi`. The variables `al` and `ah` are non-deterministically assigned true or false (*) since the initial values in the input array are unconstrained.

The **while** loop at label `L0` constrains `le` to be true if control passes into the body of the loop, as `le` is the variable corresponding to the predicate `(lo<=hi)`. The statement `lo++;` at label `L3` translates to the parallel assignment statement in the boolean program:

```
lt,le,al := ch(F,!lt), ch(lt,!lt||!le), *;
```

The `ch` function is a built-in function of boolean programs that returns true (T) if its first argument is true, false (F) if its first argument is false and second argument is true, and * (T or F) otherwise. The translation of `lo++` shows that:

	TTTT	TTTF	FTTF	FFTF	TTFT	FTFT	FFFT	TTFF	FFFF	FFTT
L0	x	x			x			x	x	
L1	x	x			x			x		
L2	x	x	x	x	x	x		x	x	
L3	x	x	x	x						
L4	x	x	x	x	x	x		x	x	
L5					x	x	x	x	x	
L6					x	x	x			
L7					x	x	x	x	x	
L8								x	x	
L9								x		
LA	x									
LB	x								x	
LC									x	

Figure 2: The reachable states of the boolean program.

- if the predicate `(lo<hi)` is false before the statement `lo++` then this predicate is false afterwards (and there is no way for this statement to make the predicate `lo<hi` true).
- if the predicate `(lo<hi)` is true before `lo++` then the predicate `(lo<=hi)` is true after; otherwise, if `(lo<hi)` is false or `(lo<=hi)` is false before then `(lo<=hi)` is false after.
- the predicate `(a[lo]<=pivot)` takes on an unknown value (*) as result of the execution of `lo++`.

The assignment statement `hi--;` at label `L6` is similarly translated. The effects of the call to the `swap` procedure at label `L9` are captured by the assignment statement `al,ah := !ah,!al;` because this call swaps the values of the elements `a[lo]` and `a[hi]`.

2.3 Reachable States in the Boolean Abstraction

We used the model checker Bebop [2] to compute the reachable states of the boolean program, as shown in Figure 2. There is a row for each of the thirteen labels in the boolean program (`L0` to `LC`) and a column for each of the ten possible valuations for the boolean variables (**lt**,**le**,**al** and **ah**).

There are 49 reachable states in the boolean program, denoted by the "x" marks in the table. This is much smaller than the total number of states, which is 130. Such sparseness is important because it rules out many states that we shouldn't even attempt to cover with tests (because they are unreachable in the boolean program, and thus in the `partition` function). Let us examine the reasons for this sparseness.

Consider the first four columns of the table. In each of these columns, the variable `al` (third bit position from left) is true. If `al` is true upon entry to the **while** loop at labels `L2` to `L4`, then

this loop iterates until `al` becomes false. This is why (with two exceptions) there is no state in which `al` is true after label `L4` in the function. The exception is due to the `swap` procedure, which makes `al` true at labels `LA` and `LB`.

Now consider the next three columns (labeled TTFT, FTFT, and FFFT). In these states, the variable `al` is false and `ah` (fourth bit position from left) is true. In these states, the first inner **while** loop does not iterate and the second inner **while** loop (labels `L5` to `L7`) will iterate until `ah` is false.

Finally, we arrive at columns labeled TTFF, FFFF, and FFTT. As can be seen in the table, due to the effect of the two inner **while** loops, the label `L8` can only be reached in one of two states (TTFF or FFFF). In the first case, the `swap` procedure will be called; in the second case, the `swap` procedure will not be called. The state FFTT is not reachable at all.

In summary, the reachable state space is sparse because of correlations between predicates in the code. This sparsity makes symbolic model checking efficient. Additionally, symbolic model checking of a boolean program has a number of advantages over directly symbolically executing a C program: (1) it can compute loop invariants over the observed predicates; (2) it is more efficient since it only observes certain aspects of the program's state. In the next section, we show how the state space of the boolean program can be used to effectively guide symbolic execution of the C code to generate test data.

2.4 Feasibility Testing and Input Generation

Our goal now is to generate test inputs that will cause each observed reachable state (in Figure 2) to be observed in an actual run of the `partition` function.

Figure 3 shows the reachable state space of the boolean program, output by the Bebop model checker. Each state is labelled `LX`:ABCD, where `LX` is the label (program counter), and A, B, C and D are the values of the boolean variables `lt`, `le`, `al`, and `ah`. Edges represent state transitions.

Solid edges in the graph are tree edges in a depth-first search (DFS) forest of the graph with roots (initial states) { `L0`:TTFT, `L0`:TTTT, `L0`:TTTF, `L0`:TTFF }. The initial states are rectangles. The dotted edges are non-tree edges (back edges, cross edges or forward edges) in the DFS forest. There are twelve leaves in the DFS forest (ovals). To cover all the states in the DFS forest requires twelve unique paths (from a rectangle to an oval), which are automatically generated as the output of the Bebop model checker.

Each of the twelve paths corresponds to a straight-line C "path" program that we automatically generated by tracing the path through the `partition` function. Let us consider one of these paths:

`L0`:TTTF → `L1`:TTTF → `L2`:TTTF → `L3`:TTTF → `L4`:TTFF → `L2`:TTFF

and its corresponding path program (see Figure 4). There are five transitions between labels in this path. The transition `L0`:TTTF → `L1`:TTTF corresponds to the expression in **while** loop at label `L0` evaluating to true. This is modeled by the statement `assume(lo<=hi)` in the path program in Figure 4. The five statements corresponding to the five transitions are presented

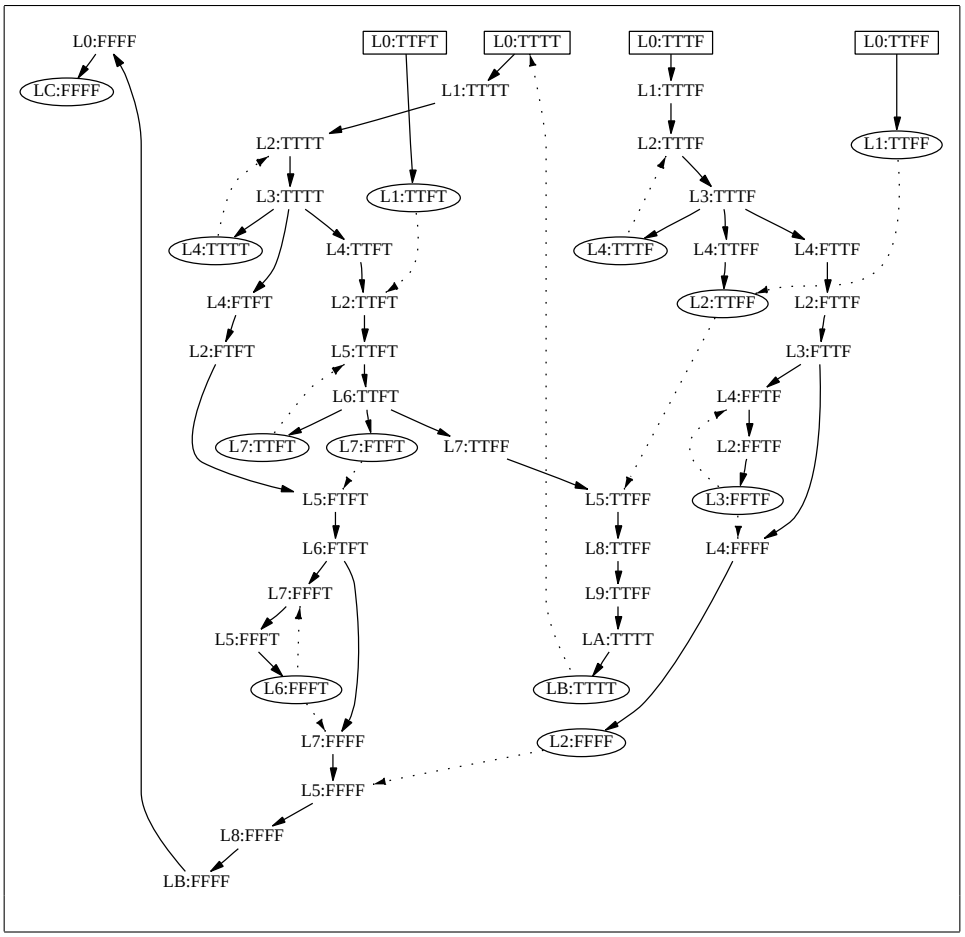

Figure 3: The state space of the boolean program and its depth-first search forest.

after the "prelude" code in Figure 4. The `assert` statement at the end of the path program asserts that the final state at label `L2` (TTFF) cannot occur.

We used CBMC [7], a bounded-model checker for C programs to determine whether or not the `assert` statement in each of the twelve path programs can fail. If it can fail then it means that all the observed states in the path are reachable in the original `partition` function and CBMC generates a counterexample which includes an input array `a[]` and array length `n` that will cause the `partition` function to visit all these observed states. (CBMC unrolls a C program to a boolean formula and uses a SAT solver to determine if the C program can fail and generate an input that would cause it to fail). If CBMC proves that the `assert` statement cannot fail then the path is infeasible. We will discusss later what can be done when infeasible paths are encountered.

For the generated path program of Figure 4, CBMC finds a counterexample and produces the input array { 1, -7, 3, 0 }.[1] Let us show that the `partition` function run on this input will

[1]CBMC actually produces very small negative input values. For our example, the negative values produced

```
partition(int a[],int n) {
  pivot = a[0];          // prelude
  lo = 1;                // prelude
  hi = n-1;              // prelude
  assume(n>2);           // prelude

  assume(lo<=hi);        // L0:TTTF -> L1:TTTF
  ;                      // L1:TTTF -> L2:TTTF
  assume(a[lo]<=pivot); // L2:TTTF -> L3:TTTF
  lo=lo+1;               // L3:TTTF -> L4:TTFF
  ;                      // L4:TTFF -> L2:TTFF

  assert(! ((lo<hi)&&(lo<=hi)&&
            !(a[lo]<=pivot)&&!(a[hi]>pivot))
         );
}
```

Figure 4: The "path" program corresponding to the path L0:TTTF → L1:TTTF → L2:TTTF → L3:TTTF → L4:TTFF → L2:TTFF.

Leaf	Input Array	CBMC Result	Bounds Failure?
L1:TTFF	{ 1, 3, 0 }	assert	
L1:TTFT	{ 0, 2, 1 }	assert	
L2:FFFF	{ 0, -7, -8 }	assert	x
L2:TTFF	{ 1, -7, 3, 0 }	assert	
L3:FFTF	{ 0, -7, -8 }	assert	x
L4:TTTF	{ 1, -7, -7, 0 }	assert	x
L4:TTTT	{ 0, -8, -8, 1 }	assert	
L6:FFFT		infeasible	
L7:FTFT	{ 0, -8, 1, 2 }	assert	
L7:TTFT	{ 0, -8, 3, 0, 1, 2 }	assert	
LB:TTTT	{ 0, -7, 1, 0, -8, 2 }	assert	
LC:FFFF	{ 0, -8, 1 }	assert	

Figure 5: Array inputs generated by CBMC for the twelve paths of the buggy C program in Figure 1(a).

cover the six reachable states in the path

L0:TTTF → L1:TTTF → L2:TTTF → L3:TTTF → L4:TTFF → L2:TTFF.

Given the input array, the initial values of (lo,hi,pivot, a[lo], a[hi]) are (lo=1,hi=3,pivot=1,a[lo]=-7,a[hi]=0) just before execution of label L0. This covers the state L0:TTTF. Since lo<=hi in this state, control will pass to labels L1 and then L2, thus covering states L1:TTTF and L2:TTTF. Since a[lo]<=pivot in the current state, control will

by CBMC are -2147483647 and -2147483648. We have substituted -7 and -8 for these two values throughout the paper for conciseness.

pass from L2 to L3, covering L3:TTTF. At this point the increment of lo takes place and the values of the five locations now are (lo=2,hi=3,pivot=1,a[lo]=3,a[hi]=0). Control passes to label L4. The expression

```
((lo<hi)&&(lo<=hi)&&!(a[lo]<=pivot)&&!(a[hi]>pivot))
```

evalutes to true and control then passes to label L2. Thus, states L4:TTFF and L2:TTFF have been covered as well.

Figure 5 shows the results of running CBMC on each of the generated path programs corresponding to the twelve paths to leaf vertices in the DFS forest. If the column **CBMC result** contains "assert", this means that CBMC found a counterexample that caused the final `assert` statement to fail. If the column contains "infeasible" it means that CBMC proved that the final `assert` cannot fail. CBMC finds that eleven of the twelve paths generated from the boolean program are feasible in the source program.

The infeasible path is due to the fact that the state L6:FFFT is not reachable in the `partition` function, but is reachable in the boolean program. The path is infeasible in the `partition` function because the `hi` variable has been decremented so that its value is less than that of the `lo` variable. Since the `partition` function maintains the invariant that all array elements with index less than `lo` have value less than or equal to `pivot`, the value of `a[hi]` must be less than or equal to pivot. However, the state L6:FFFT requires that `a[hi]>pivot` is true at the end of the path. The reason the path is feasible in the boolean program is that our four chosen predicates do not track the values of the array elements that are below the index `lo` or are above the index `hi`.

Three of the twelve feasible paths exhibit a bounds violation, namely those with leaf states L2:FFFF, L3:FFTF and L4:TTTF. For example, the path with leaf state L2:FFFF and the path with leaf state L3:FFTF both have the input array { 0, -7, -8 }, which will cause the `partition` function to advance the `lo` index beyond the upper bound of the array.[2]

2.5 Analysis of the Corrected Partition Function

We fixed the `partition` function to eliminate the two array bounds violations (first and second inner **while** loops) and re-ran our entire process on the fixed function. The end result of this analysis is shown in Figure 6. The results for ten of the leaf states (L1:TTFF, L1:TTFT, L2:FFFF, L2:TTFF, L4:TTTF, L4:TTTT, L7:FTFT, L7:TTFT, LB:TTTT, LC:FFFF) are exactly the same as before.

The leaf states L3:FFTF and L6:FFFT, which were reachable in the buggy boolean program, are no longer reachable in the fixed boolean program. The first state corresponded to a bounds violation (which has been eliminated) and the second state was unreachable in the buggy `partition` function.

However, two new leaf states have been found: LC:FFFT and LC:FFTF. The first state is unreachable (for the same reason that the state L6:FFFT was unreachable in the buggy program). The second state is a new state that is reachable as a result of the bug fix.

[2]The inquisitive reader may wonder how two different paths can generate the same input array. This is because the value of `a[lo]` is undefined when the variable `lo` steps beyond the bounds of the array { 0, -7, -8 }. Thus, the predicate `a[lo]<=pivot` could be true or false, giving rise to two different paths.

Leaf	Input Array	CBMC Result
L1:TTFF	{ 1, 3, 0 }	assert
L1:TTFT	{ 0, 2, 1 }	assert
L2:FFFF	{ 0, -7, -8 }	assert
L2:TTFF	{ 1, -7, 3, 0 }	assert
L4:TTTF	{ 1, -7, -7, 0 }	assert
L4:TTTT	{ 0, -8, -8, 1 }	assert
L7:FTFT	{ 0, -8, 1, 2 }	assert
L7:TTFT	{ 0, -8, 3, 0, 1, 2 }	assert
LB:TTTT	{ 0, -7, 1, 0, -8, 2 }	assert
LC:FFFF	{ 0, -8, 1 }	assert
LC:FFFT		infeasible
LC:FFTF	{ 0, -7, -8 }	assert

Figure 6: Array inputs generated by CBMC for the corrected program (no array bounds violations).

3 Discussion

3.1 Predicate Selection

Our approach to test generation is parameterized by the set of predicates E, which define the precision of the boolean abstraction $BP(P, E)$. The more precise this abstraction, the more likely that the paths generated by the Bebop model checker ($P(S_E)$) will be feasible paths ($F(S_E)$) in the source program P. This led us to propose the ratio $|F(S_E)|/|P(S_E)|$, which measures the precision of the boolean program abstraction by its ability to find feasible paths in the source program.

To illustrate this concept, consider using our test generation process on the `partition` function with no predicates. In such a case, every path through the function would be considered by the model checker, including many infeasible paths. For example, the model checker would output the (infeasible) path in which the outermost **while** loop does not iterate. However, as we can see from the source program, the outermost **while** loop must iterate at least once.

What exactly does a $|F(S_E)|/|P(S_E)|$ ratio of 1.0 signify? It says that no more tests are needed in order to cover the observable states in S_E. However, it doesn't mean that all interesting observations about a piece of code have been made. In our running example, the specification for the `partition` function illustrates other observations we could make. This specification states that at label `L1` the following loop invariant holds:

for all `i`, `0<=i<lo`, `(a[i]<=pivot)` and for all `j`, `hi<j<n`, `(pivot<a[j])`

Our basic point is not new: generating test data based solely on the code is never sufficient– one must also consider the specification of what the code is supposed to as well. [10]

If a path in p_s generated by the model checker is infeasible in the C program, it may be that the state s at the end of p_s is unreachable in the C program or it may be that there is a feasible

path to s but the predicates in E were insufficient to guide the model checker to this path. In this case, there are several alternatives: involve the programmer in the process to either add predicates to E to guide the model checker to a feasible path to s or assert that the state is not reachable; use an automated tool such as SLAM to try to prove that s is reachable/unreachable in the C program, thereby generating more predicates.

3.2 Partial Programs and "Ping-Pong" Analysis

It often will be the case that the source code of a unit is compiled and linked against existing (binary) libraries for which no source code is available. We propose a simple idea for dealing with such code: use our test generation technique on the code for which we have source, run the tests and observe the effect of the binary code on the observation predicates.

In our running example, suppose that the `swap` function is only available as a binary. In this case, it will not be possible to construct an accurate boolean program model of the `partition` function at the call to the `swap` procedure. At the call to `swap`, every predicate that can be potentially affected must be invalidated.

However, since every other program point in the `partition` function can be abstracted precisely, we can still use our process to generate inputs that will cause the `swap` function to execute. For such test inputs, we observe the affect of `swap` on the four observation predicates and incorporate these observations into the boolean program abstraction. The idea is to let our analysis "ping-pong" between an "abstract" phase of predicate abstraction, model checking and symbolic execution and a "concrete" phase of execution on generated inputs. Observations of the values of the predicates E during concrete execution can be fed back into the abstract execution phase to refine the knowledge about the behavior of library code.

3.3 The Small Scope Hypothesis

The "small scope hypothesis" of testing is that a high percentage of the bugs in a system can be found by exhaustively checking the program on inputs of a small size. If one believes this hypothesis then a central question is "how large should we choose 'small' to be?" Our process helps to generate such small inputs that cover a set of observed states. In our example, the size of the input arrays range from length three to six. The observations E place constraints on the size of the input needed to cover the observable states S_E. In effect, our process can help determine how large "small" should be.

4 Related Work

The idea of using paths and symbolic execution of paths to generate tests has a long and rich history going back to the mid-1970's. [5, 15, 8, 22] and continuing to the present day [17, 11, 13]

The major contribution of our work over previous efforts is to guide test generation using predicate abstraction and model checking. We use the automatically-created boolean program abstraction to guide the search for feasible paths in a C program. This abstraction is based on observations (predicates) over the state space of the C program. These observations can be

taken directly from the code (as they appear in conditionals) or provided by programmers or testers. In fact, this parameterization of test generation via predicates makes it possible for the programmer or tester to increase the level of testing thoroughness through the addition of new observations. Finally, the boolean program abstraction provides a denominator ($P(S_E)$) by which we can assess the test generation effectiveness of a set of predicates E.

A classic problem in path-based symbolic execution is the selection of program paths. One way to guide the search for feasible paths is to execute the program symbolically along all paths, while guiding the exploration to achieve high code coverage. Clearly, it is not possible to symbolically execute all paths, so the search must be cut off at some point. Often, tools will simply analyze loops through one or two iterations. [6] Another way to limit the search is to bound the size of the input domain (say, to consider arrays of at most length three) [16], or to bound the maximum path length that will be considered, as done in bounded model checking. [7] An experiment by Yates and Malevris provided evidence that the likelihood that a path is feasible decreases as the number of predicates in the path increases. [23] This led them to use shortest-path algorithms to find a set of paths that covers all branches in a function.

In contrast to all these heuristics, our technique uses the set of input predicates E to bound the set of paths that will be used to generate test data for a program P. The predicates induce a boolean abstraction $BP(P, E)$ which guides the selection of paths, with the goal of covering the set of observable states S_E.

Other approaches to test generation rely on dynamic schemes. Given an existing test t, Korel's "goal-oriented" approach seeks to perturb t to a test t' cover a particular statement, using function minimization techniques. [18] The potential benefit of Korel's approach is that it is dynamic and has an accurate view of memory and flow dependences. Ideas from his approach may be applicable in a "ping-pong" analysis. The downside of his approach is that test t may be very far away from a suitable test t'.

Another dynamic approach to test generation is found in the Korat tool. [4]. This tool uses a function's precondition on its input to automatically generate all (nonisomorphic) test cases up to a given small size. It exhaustively explores the input space of the precondition and prunes large portions of the search space by monitoring the execution of the precondition. For an example such as the `partition` function that has no constraints on its input, the Korat method will not work very well. Furthermore, it requires the user to supply a bound on the input size whereas our technique infers the input size.

Random or fuzz testing is another popular technique for unit testing. [9], [20], [21] For a simple example such as the `partition` function, random generation of arrays would probably perform quite well. It will be interesting to compare random testing with our technique for more complex examples.

Of course, if a designer provides a specification of the expected behavior of a software system, this specification can be used to drive test generation as well. State-based test generation from manually-provided models has been widely studied and applied. [19] Such techniques are complementary to our code-based approach, which creates abstract models from code automatically.

5 Conclusion

We have presented a process for using predicate abstraction and model checking to guide test case generation via symbolic execution, and have applied the process to a small example. Clearly, much more needs to be done to judge whether or not this process will scale and be useful. Towards that end, we plan to implement the process fully for Microsoft's intermediate language (MSIL) that is the bytecode representation now targeted by Microsoft compilers for C# and Visual Basic. Our plan is to create a unit testing tool that will automate test generation for MSIL and investigate the "ping-pong" approach outlined in Section 3.2 for incorporating results from test executions back into symbolic analysis.

Acknowledgements

Thanks to Daniel Kroening for his help with the CBMC model checker. Thanks also to Byron Cook and Vladimir Levin for their comments on drafts of this paper.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 123–133. ACM, 2002.

[5] R. Boyer, B. Elspas, and K. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, 1975.

[6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[7] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Design Automation Conference*, pages 368–371, 2003.

[8] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[9] J. W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 179–183. IEEE, 1981.

[10] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1976.

[11] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62. ACM, 1998.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[13] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *FSE 98: Foundations of Software Engineering*. ACM, 1998.

[14] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proceedings of Automated Software Engineering*, pages 219–222, 2000.

[15] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2:208–215, 1976.

[16] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.

[17] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 95–107. ACM, 1994.

[18] B. Korel. Dynamic method of software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[19] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[20] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[21] S. Ntafos. On random and partition testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 42–48. ACM, 1998.

[22] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300.

[23] D. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification*, pages 48–54. ACM, 1989.