A Theory of Predicate-Complete Test Coverage and Generation*

Thomas Ball
tball@microsoft.com
Microsoft Research

April 23, 2004

Technical Report
MSR-TR-2004-28

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

A Theory of Predicate-Complete Test Coverage and Generation*

Thomas Ball
tball@microsoft.com
Microsoft Research

Abstract

Consider a program with m statements and n predicates, where the predicates are derived from the conditional statements and assertions in a program, as well as from implicit run-time safety checks. An observable state is an evaluation of the n predicates under some state at a program statement.

The goal of *predicate-complete* testing (PCT) is to cover every reachable observable state (at most $m \times 2^n$ of them) in a program. PCT coverage is a new form of coverage motivated by the observation that certain errors in a program only can be exposed by considering the complex dependences between the predicates in a program and the statements whose execution they control. PCT coverage subsumes many existing control-flow coverage criteria and is incomparable to path coverage.

To support the generation of tests to achieve high PCT coverage, we show how to define an upper bound U and lower bound L to the (unknown) set of reachable observable states R. These bounds are constructed automatically using Boolean (predicate) abstraction over modal transition systems and can be used to guide test generation via symbolic execution. We define a static coverage metric as $|L|/|U|$, which measures the ability of the Boolean abstraction to achieve high PCT coverage. Finally we show how to increase this ratio by the addition of new predicates.

1 Introduction

Control-flow-based test generation generally has as its goal to cover all the statements or branches in a program. There are various control-flow adequacy criteria that go beyond branch coverage, such as multiple condition coverage, the ultimate of which is path coverage. Errors that go undetected in the face of 100% statement or branch coverage may be due to complex correlations between the predicates (that control the execution of statements) and the statements (that affect the value of these predicates) of a program. However, paths are notoriously difficult to work with as a coverage metric because there are an unbounded number of them in programs with loops, which characterizes most interesting programs in existence.

So, we seek an alternative to path coverage that has its "exhaustive" quality but induces a finite (rather than infinite) state space. We start with a fixed notation for atomic predicates (not containing Boolean connectives), taken from the relevant programming language. 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. Consider a program with m statements and n predicates. These predicates can be drawn from the conditional statements and assertions in a program, as well as from implicit run-time safety checks (for checking for array bounds violations or divide-by-zero errors, for example) or from automated analysis or the programmer. An *observable state* is an evaluation of the n predicates under some program state at a statement. While the set of states in a program is unbounded, the size of the set of observable states (S) is at most $(m \times 2^n)$.

We define the (theoretic) goal of *predicate-complete* testing (PCT) to be to cover all *reachable* observable states. The n predicates represent all the case-splits on the input that the programmer has identified.[1] In the limit, each of the m statements may have different behavior in each of the 2^n possible observable states, and so should be tested in each of these states. We show that PCT coverage subsumes traditional coverage metrics such as statement, branch and multiple condition coverage and that PCT coverage is incomparable to path coverage. PCT groups paths ending at a statement s into equivalence classes based on the observable states the paths induce at s.

Control-flow coverage metrics result from dividing a dynamic measure (for example, the number of statements executed by a test) into a static measure (for example, the number of statements in a program). Clearly, such a metric also can be defined for observable states. However, the choice of $(m \times 2^n)$ as a denominator will not do, as we expect many of the $(m \times 2^n)$ states to be unreachable. (Statement coverage does not suffer greatly from this problem because most statements are reachable). For example, if the set of predicates contains $(x = 0)$ and $(x = 1)$ then not all combinations are possible. Furthermore, invariants established by the program will further cut down the number of reachable observable states.

Thus, we desire a way to define a better denominator for PCT coverage. The main result of this paper is a way to overapproximate and underapproximate the set of reach-

* A revised version of this paper will appear in the proceedings of the 2004 Third International Symposium on Formal Methods for Components and Objects (FMCO 2004), Leiden, The Netherlands, November 2-5.

[1] Of course, the programmer may have missed certain cases– specification-based testing would need to be used to determine the absence of such case-splits.

able observable states (R) using the theory of modal transition systems and Boolean abstraction. The Boolean abstraction of a program with respect to its n predicates is a non-deterministic program, whereas the original concrete program is deterministic. We show how reachability analysis of this abstract program yields an upper bound U for R ($R \subseteq U$) as well as a lower bound L for R ($L \subseteq R$). The set U is an *overapproximation* of R: any state outside U is not a reachable observable state and need not (indeed, cannot) be tested. This set U provides a better denominator than ($m \times 2^n$). Conversely, the set L is an *underapproximation* of R: any state in L *must* be a reachable observable state. Any state in L must be testable.

The reachability status of states in $U - L$ is unknown. If a set of tests doesn't cover some states in L, one should first try to cover these states. We show how to use L to guide symbolic path-based test generation to cover the untested states in L. After covering L, one should work on bringing the (static) ratio $|L|/|U|$ closer to one by refining the Boolean abstraction through the introduction of additional predicates.

This paper is organized as follows. Section 2 compares predicate-complete test coverage to other forms of control-flow coverage. Section 3 gives an example to illustrate the idea of upper and lower bounds to the set of reachable observable states. Section 4 precisely defines the system of abstraction we will use to compute the upper and lower bounds. Section 5 gives algorithms to compute these upper and lower bounds. Section 6 presents an algorithm that uses the lower bound to guide test generation. Section 7 shows how to refine the upper and lower bounds in the running example to increase PCT coverage. Section 8 discusses some of the implications of our results. Section 9 reviews related work and Section 10 concludes the paper.

2 A Characterization of Predicate-complete Test Coverage

This section compares PCT coverage with other forms of control-flow coverage. In this comparison, we decompose complex predicates into atomic predicates. So, the program fragment

```
L1: if ((x<0)||(y<0)) S else T
```

contains two branches corresponding to the atomic predicates `(x<0)` and `(y<0)`. Based on this decomposition, the concepts of branches, atomic predicates and conditions are equivalent.

To recap, complete PCT coverage means that each reachable observable state of a program is covered by a test. This implies that each (executable) statement is executed at least once, so PCT subsumes statement coverage. PCT coverage requires that each predicate be tested so as to evaluate to both true and false (of course this may not be possible for unsatisfiable predicates such as `(x!=x)`), so it subsumes branch coverage. PCT clearly also subsumes multiple condition coverage and its variants. Considering the program fragment given above, multiple condition coverage requires every possible Boolean combination of `(x<0)` and `(y<0)` to be tested at `L1`, which seems similar to PCT. But now, consider the sequencing of two **if** statements:

```
L2: if (A || B) S else T
L3: if (C || D) U else V
```

```
                  // reachable observable states
L0:  y = 0;       // (L0,x<0)     (L0,!(x<0))
L1:  if (x<0)     // (L1,x<0)     (L1,!(x<0))
L2:    skip;      // (L2,x<0)
     else         //
L3:    x = -2;    //              (L3,!(x<0))
L4:  x = x + 1;   // (L4,x<0)
L5:  if (x<0)     // (L5,x<0)     (L5,!(x<0))
L6:    y = 1;     // (L6,x<0)
```

(a)

```
L1:  if (p)
L2:    if (q)
L3:      x=0;
L4:  B;
```

(a)

Figure 1: (a) A program that shows it is possible to attain full PCT coverage without covering all feasible paths. (b) A program that that shows it is possible to cover all feasible paths without attaining full PCT coverage.

PCT requires that every Boolean combination over the set { `A`, `B`, `C`, `D` } be tested at every statement in the program (six in this case, the two **if** statements and the four statements `S`, `T`, `U` and `V`). Multiple condition coverage only requires that every Boolean combination over { `A`, `B` } be tested at `L2` and that every that every Boolean combination over { `C`, `D` } be tested at `L3`. Similarly, predicate-based test generation [Tai03, Tai96] focuses on testing predicates in a program. It considers correlations between predicates that appear in a single conditional statement but does not consider correlations between predicates that appear in different conditional statements, as does PCT.

Of course, we can view paths as possible logical combinations of predicates, so it is natural to ask how PCT relates to path coverage. Since a program with n predicates can have at most 2^n paths, it seems like PCT might have the ability to explore more behaviors (as it may explore $m \times 2^n$ states in the limit). In fact, we show PCT and path coverage are incomparable, even for loop-free programs.

The program in Figure 1(a) shows that it is possible to cover all reachable observable states in a (loop-free) program without covering all feasible paths. In this program, we assume that the uninitialized variable `x` can take on any initial (integer) value. The reachable observable states of this program are shown in the comments to the right of the program. The set of tests { $x \to -1$, $x \to 1$ } covers all these states. The test { $x \to -1$ } covers the observable states

{ `(L0,x<0)`, `(L1,x<0)`, `(L2,x<0)`, `(L4,x<0)`, `(L5,!(x<0))` }

via the path (`L0`,`L1`,`L2`,`L4`,`L5`), while the test { $x \to 1$ } covers the observable states

{ `(L0,!(x<0))`, `(L1,!(x<0))`, `(L3,!(x<0))`, `(L4,x<0)`, `(L5,x<0)`, `(L6,x<0)` }

via the path (`L0`,`L1`,`L3`,`L4`,`L5`,`L6`). However, this set of tests does not cover the feasible path (`L0`,`L1`,`L2`,`L4`,`L5`,`L6`), which is covered by the test { $x \to -2$ }.

Because of the assignment statement "`x = -2;`", the set of reachable observable states at label `L4` (namely (`L4`,`x<0`)) cannot distinguish whether the executed path to `L4` traversed the **then** or **else** branch of the initial **if** statement.

```
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 := (!lt ? F : *), lt, *;
L4:     skip;
      od
L5:   while (ah) do
L6:     lt,le,ah := (!lt ? F : *), lt, *;
L7:     skip;
      od
L8:   if (lt) then
L9:     al,ah := !ah,!al;
LA:     skip;
      fi
LB:   skip;
    od
LC: skip;
end
```

(b)

Figure 2: (a) The `partition` function and (b) its Boolean program.

While PCT can track many correlations, assignment statements such as the one above can cause PCT to lose track of correlations captured by path coverage.

In this example, if we add the predicate `(x==-2)` to the set of observed predicates then PCT coverage is equivalent to path coverage, as PCT coverage will require the test $\{ x \rightarrow -2 \}$ in order to cover the reachable state (L2,x==-2). It is an open question whether we can always find a minimal set of predicates for which PCT coverage implies path coverage (or decide that only infinitely many predicates will do).[2]

Figure 1(b) shows that it is possible to cover all feasible paths in a (loop-free) program without covering all reachable observable states. The program has three feasible paths: (L1,L2,L3,L4), (L1,L2,L4) and (L1,L4). However, a test set of size three that covers these paths clearly will miss either the observable state (L4,!p&&q) or (L4,!p&&!q).

In summary, PCT coverage is a new type of coverage criteria that subsumes statement, branch, multiple condition and predicate coverage. PCT has similarities to path coverage but is strictly incomparable, as the above examples demonstrate. In Section 9 we compare PCT coverage to several other control-flow coverage criteria.

3 Example

This section demonstrates upper and lower bounds to the reachable observable states of a small function. Figure 2(a) presents a (buggy) example of QuickSort's `partition` function, a classic example that has been used to study test generation [BEL75]. 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 is an array bound check missing in the code that can lead to an array bounds error: the check at the **while** loop at label L2 should be `(lo<=hi && a[lo]<=pivot)`.[3] This error only can be uncovered by executing the statement "`lo++`" at label L3 at least twice.

There are thirteen labels in the `partition` function (L0-LC), but an unbounded number of paths. 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 conditional guards 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.

These correlations reduce the possible observable state space from $13 * 16 = 208$ to $13 * 10 = 130$.

[2] Thanks to Orna Kupferman for suggesting this question.

[3] The loop at L5 cannot decrement `hi` to take a value less than zero because the value of variable `pivot` is fixed to be the value of `a[0]`. One could argue that one would want to put a bounds check in anyway.

	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 3: The reachable states of the Boolean abstraction of the `partition` function with respect to the four predicates in that function. The Boolean values in each column correspond to the variables **lt**, **le**, **al** and **ah**, respectively.

3.1 Boolean Abstraction

Figure 2(b) shows the Boolean abstraction of the `partition` function with respect to the four observed predicates, encoded as a Boolean program [BR00]. This program can be automatically constructed using software predicate abstraction technology [BMMR01]. The Boolean program has one variable (`lt`, `le`, `al`, `ah`) corresponding to each predicate. Statements in the Boolean program conservatively update each Boolean variable to track the value of its corresponding predicate. The `enforce` statement is a global assumption that rules out the six infeasible valuations mentioned above.

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 nondeterministically 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 := (!lt ? F : *), lt, *;
```

The translation of "`lo++`" shows that:

- if the predicate `(lo<hi)` is false before the statement "`lo++`" then this predicate is false afterwards; otherwise, the predicate takes on an unknown value (*);
- if the predicate `(lo<hi)` is true before "`lo++`" then the predicate `(lo<=hi)` is true after; otherwise, if `(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]`.

The Boolean program is an abstraction of the C program in the following sense: any state transition $c \rightarrow c'$ in the C program is matched by a corresponding transition $a \rightarrow a'$ in the Boolean program, where a is the abstract state corresponding to c and a' is the abstract state corresponding to c'. However, there may be state transitions in the Boolean program that are not matched by transitions in the C program. In this sense, the Boolean program has more behaviors than the C program (the Boolean program overapproximates the behaviors of the C program).

3.2 Upper and Lower Bounds

Figure 3 shows the reachable states of the Boolean program (computed using the Bebop model checker [BR00]). 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. These 49 states represent the upper bound U to the set of reachable observable states in the `partition` function. States outside U cannot be tested because they are unreachable in the Boolean program, and thus in the `partition` function.

Figure 4 shows the reachable state space of the Boolean program as a graph. Each of the 49 states is uniquely labeled `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`. The (four) initial states are denoted by ovals. Each edge in the graph represents a transition between two reachable states of the Boolean program. Informally, a solid edge represents a transition that *must* occur in the `partition` function, while a dotted edge represents a transition that *may* occur in the `partition` function. We will formally define these transitions later. Consider the initial state `L0`:TTTT. This abstract state corresponds to all concrete states that satisfy the expression:

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

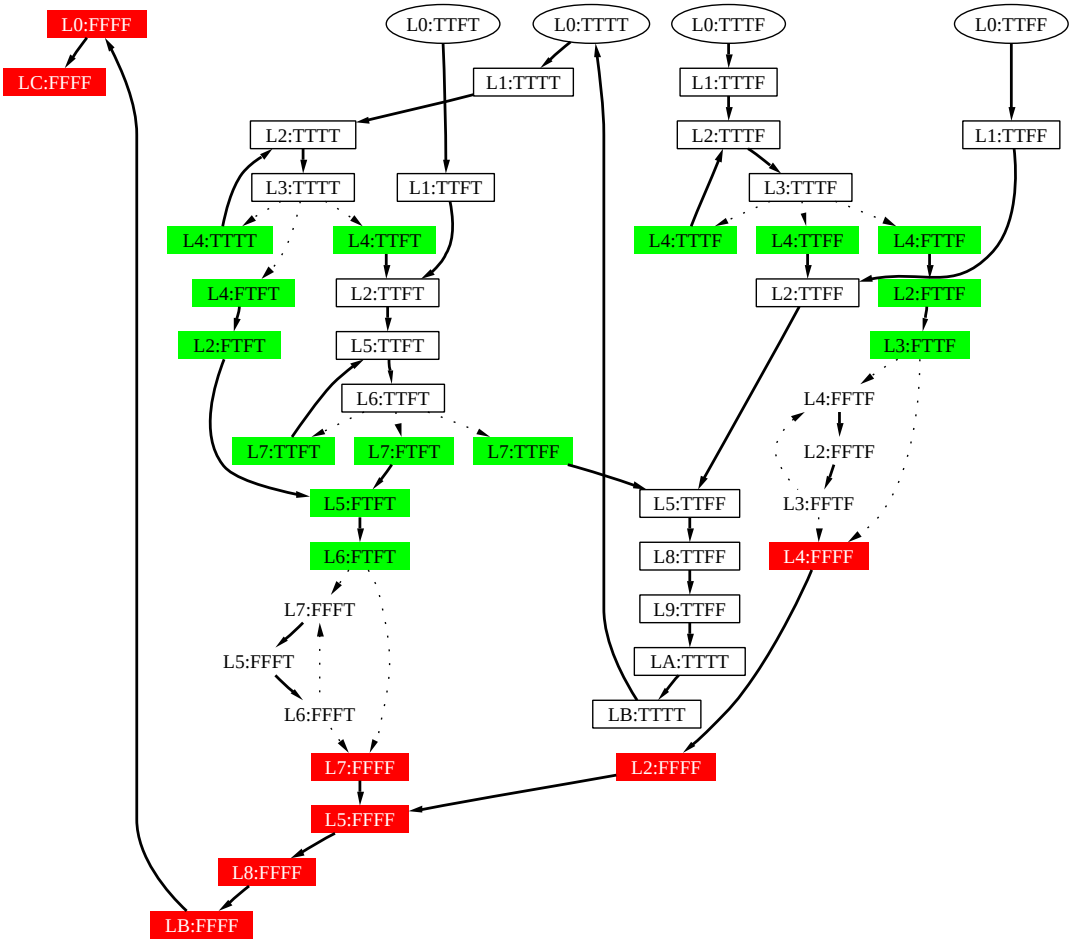

Figure 4: The reachable state space of the Boolean program (representing the upper bound U to the set of reachable observable states of the original program). The ovals represent the initial states $I_A = \{$ `L0`:TTFT, `L0`:TTTT, `L0`:TTTF, `L0`:TTFF $\}$. The ovals and rectangles comprise the lower bound L, while the plaintext nodes represent the set $U - L$.

States satisfying this condition will cause the body of the outer **while** loop (label `L1`) and the body of the first inner **while** loop (label `L3`) to execute. This is reflected in the state space by the sequence of *must*-transitions `L0`:TTTT $\rightarrow$ `L1`:TTTT $\rightarrow$ `L2`:TTTT $\rightarrow$ `L3`:TTTT.

The set of nodes in Figure 4 represent the states that comprise the upper bound U ($|U| = 49$). The ellipses and rectangles comprise the lower bound L ($|L| = 43$).[4] The nodes rendered in plaintext represent those states that are in $U - L$. The exact reachability status of the concrete states corresponding to these abstract states is in question.

The ratio $|L|/|U|$ measures the ability of the abstraction to guide test generation to cover the observable states of a program. In this example, the ratio is 43/49. If we achieve a ratio of 1.0, then we have precisely characterized the set of reachable observable states. We wish to increase the ratio $|L|/|U|$ through the process of abstraction refinement. As we will see, it is possible to reach a ratio of 1.0 for the `partition` function through the addition of three predicates.

[4] We will explain the coloring of the rectangles later.

4 Formalizing Abstraction

In this section, we define the concepts of concrete and abstract transition systems that we will use to compute the upper and lower bounds, U and L, to the set of reachable observable states R of a program.

4.1 Concrete Transition Systems

We represent a deterministic sequential program by a concrete transition system (CTS) as follows:

Definition 4.1: (*Concrete Transition System*). A concrete transition system is a triple $(S_C, I_C, \longrightarrow)$ where S_C and I_C are non-empty sets of states and $\longrightarrow \subseteq S_C \times S_C$ is a transition relation satisfying the following constraints:

- $S_C = \{halt,\ error\} \cup T_C$;
- $I_C \subseteq T_C$ is the set of initial states;
- $\forall s_c \in T_C,\ |\{s'_c \in S_C \mid s_c \longrightarrow s'_c\}| = 1$

There are two distinguished end states, *halt* and *error*, which correspond to execution terminating normally and going wrong, respectively. These two states have no successor states. All other states have exactly one successor. Thus, a CTS models a program as a set of traces.

4.2 Abstract Transition Systems

Modal Transition Systems (MTSs) [GR03] are a formalism for reasoning about partially defined systems that we will use to model (Boolean) abstractions of CTSs. We generalize modal transition systems to tri-modal transition systems (TTSs) as follows:

Definition 4.2: (*Tri-Modal Transition System*). A TTS is a tuple $(S, \overset{may}{\longrightarrow}, \overset{must^+}{\longrightarrow}, \overset{must^-}{\longrightarrow})$ where S is a nonempty set of states and $\overset{may}{\longrightarrow} \subseteq S \times S$, $\overset{must^+}{\longrightarrow} \subseteq S \times S$ and $\overset{must^-}{\longrightarrow} \subseteq S \times S$ are transition relations such that $\overset{must^+}{\longrightarrow} \subseteq \overset{may}{\longrightarrow}$ and $\overset{must^-}{\longrightarrow} \subseteq \overset{may}{\longrightarrow}$.

A total-onto abstraction relation[5] ρ induces an abstract TTS M_A from a CTS M_C as follows [God03]:

Definition 4.3: (*Precise Abstraction Construction*). Let $M_C = (S_C, I_C, \longrightarrow)$ be a CTS. Let S_A be a set of abstract states and ρ be a total-onto abstraction relation over pairs of states in $S_C \times S_A$. Let $match^+$ and $match^-$ relate states in $S_C \times S_A$ as follows:

$$match^+(s_c, s'_a) = \exists (s'_c, s'_a) \in \rho : s_c \longrightarrow s'_c$$

$$match^-(s'_c, s_a) = \exists (s_c, s_a) \in \rho : s_c \longrightarrow s'_c$$

A TTS $M_A = (S_A, \overset{may}{\longrightarrow}_A, \overset{must^+}{\longrightarrow}_A, \overset{must^-}{\longrightarrow}_A)$ is constructed from M_C, S_A and ρ as follows:

(a) $s_a \overset{may}{\longrightarrow}_A s'_a$ **iff** $\exists (s_c, s_a) \in \rho : match^+(s_c, s'_a)$;

(b) $s_a \overset{must^+}{\longrightarrow}_A s'_a$ **iff** $\forall (s_c, s_a) \in \rho : match^+(s_c, s'_a)$;

(c) $s_a \overset{must^-}{\longrightarrow}_A s'_a$ **iff** $\forall (s'_c, s'_a) \in \rho : match^-(s'_c, s_a)$.

It is easy to see that the definition of M_A satisfies the constraints of a TTS, namely that $\overset{must^+}{\longrightarrow}_A \subseteq \overset{may}{\longrightarrow}_A$ and $\overset{must^-}{\longrightarrow}_A \subseteq \overset{may}{\longrightarrow}_A$.

We have emphasized the "**iff**" (if-and-only-if) text to make a point that we assume it is possible to create a most precise abstract TTS M_A from a given CTS M_C. In general, this assumption does not hold for infinite-state M_C. It does hold for the `partition` function and other code examples we consider here.

Figure 5 illustrates the three types of transitions in a TTS M_A constructed from a CTS M_C via the above definition. In this figure, the grey nodes represent states in S_C and edges between these nodes represent transitions in $\longrightarrow$. The dotted ovals around the nodes represent an abstract state that these concrete states map to under the abstraction relation ρ. Let us examine the four cases in Figure 5:

[5] A total-onto relation over $D \times E$ contains at least one pair (d, e), $e \in E$, for each element $d \in D$ (it is total) and at least one pair (d', e'), $d' \in D$, for each element $e' \in E'$ (it is onto).

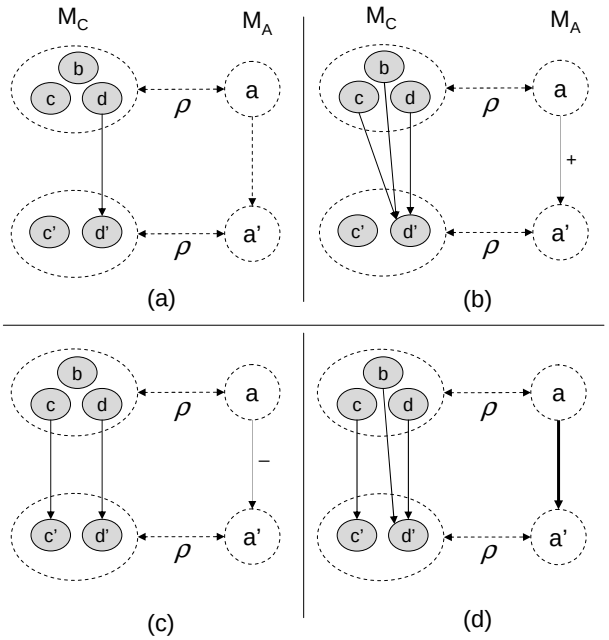

Figure 5: Illustrations of (a) a *may*-transition; (b) a $must^+$-transition; (c) a $must^-$-transition; (d) a transition that is a $must^+$-transition and a $must^-$-transition.

- Case (a) shows a transition $a \overset{may}{\longrightarrow}_A a'$. *May*-transitions are depicted as dashed edges. This transition exists because concrete state d maps to a (under ρ) and transitions to d' via $d \longrightarrow d'$, where d' maps to a'. Thus, $match^+(d, a')$ holds. Note, however that $match^+(c, a')$ does not hold, nor does $match^-(c', a)$. Therefore, in this case, there is no transition $a \overset{must^+}{\longrightarrow}_A a'$ or $a \overset{must^-}{\longrightarrow}_A a'$.

- Case (b) shows a transition $a \overset{must^+}{\longrightarrow}_A a'$, depicted as a solid edge with a "+" label. This transition exists because for all states $x \in \{b, c, d\}$ (mapping to a under ρ), $match^+(x, a')$ holds. This is due to the existence of the transitions $b \longrightarrow d'$, $c \longrightarrow d'$ and $d \longrightarrow d'$. That is, $must^+$-transitions identify a *total* relation between sets of concrete states corresponding to a and a'. Note that $match^-(c', a)$ does not hold, so there is no transition $a \overset{must^-}{\longrightarrow}_A a'$.

- Case (c) shows a transition $a \overset{must^-}{\longrightarrow}_A a'$, which exists because for all states $x' \in \{c', d'\}$ (mapping to a' under ρ), $match^-(x', a)$ holds. That is, $must^-$-transitions identify an *onto* relation between sets of concrete states corresponding to a and a'. These transitions are depicted as solid edges with "-" labels.

- Case (d) shows the case in which there are both transitions $a \overset{must^+}{\longrightarrow}_A a'$ and $a \overset{must^-}{\longrightarrow}_A a'$. Let $a \overset{must^\#}{\longrightarrow}_A a'$ denote the fact that $a \overset{must^+}{\longrightarrow}_A a'$ and $a \overset{must^-}{\longrightarrow}_A a'$. These transitions are depicted as bold edges.

In the example, of Figure 4, we have only $\overset{must^\#}{\longrightarrow}_A$-transitions (bold edges) and *may*-transitions (dotted edges).

4.3 Predicate Abstraction

Predicate abstraction maps a (potentially infinite-state) CTS into a finite-state TTS via a finite set of quantifier-free formulas of first-order logic $\Phi = \{\phi_1, \cdots, \phi_n\}$. A bit vector b of length n ($b = b_1 \cdots b_n$, $b_i \in \{0,1\}$) defines an abstract state whose corresponding concrete states are those satisfying the conjunction $\langle b, \Phi\rangle = (l_1 \wedge \cdots \wedge l_n)$ where $l_i = \phi_i$ if $b_i = 1$ and $l_i = \neg\phi_i$ if $b_i = 0$. We write $s \models \langle b, \Phi\rangle$ to denote that $\langle b, \Phi\rangle$ holds in state s.

Definition 4.4: (*Predicate Abstraction of a CTS*). Given a CTS $M_C = (S_C, I_C, \longrightarrow)$ and a set of predicates $\Phi = \{\phi_1, \cdots, \phi_n\}$, predicate abstraction defines the total-onto abstraction relation ρ and the set of abstract states S_A:

- $\rho \in (S_C, \{0,1\}^n)$, where $(s,b) \in \rho \iff s \models \langle b, \Phi\rangle$
- $S_A = \{b \in \{0,1\}^n \mid \exists (s,b) \in \rho\}$

which define the finite-state abstract TTS $M_A = (S_A, \xrightarrow{may}_A, \xrightarrow{must^+}_A, \xrightarrow{must^-}_A)$ (per Definition 4.3). We assume that S_A contains abstract states $halt_A$ and $error_A$ that are in a one-to-one relationship with their counterparts *halt* and *error* from S_C.

Algorithms for computing the *may*- and $must^+$-transitions of a predicate abstraction of an MTS are given by Godefroid, Huth and Jagadeesan [GHJ01]. Computation of the $must^-$-transitions can be done in a similar fashion Computation of the most precise abstract transitions is undecidable, in general. As usual, we assume the existence of a complete theorem prover that permits the computation of the most precise abstract transitions.

We review the basic idea here. Let $WP(s,e)$ be the weakest pre-condition of a statement s with respect to expression e and let $SP(s,e)$ be the strongest post-condition of s with respect to e [Gri81]. (For any state c_1 satisfying $WP(s,e)$ the execution of s from c_1 results in a state c_2 satisfying e. For any state c_1 satisfying e the execution of s from c_1 results in a state c_2 satisfying $SP(s,e)$). Let P_1 and P_2 be the concretization of two bit vectors b_1 and b_2 (i.e., $P_1 = \langle b_1, \Phi\rangle$ and $P_2 = \langle b_2, \Phi\rangle$). Statement s induces a *may*-transition from b_1 to b_2 if $\exists V : P_1 \wedge WP(s, P_2)$, where V is the set of free variables in the quantified expression. Statement s induces a $must^+$-transition from b_1 to b_2 if $\forall V.P_1 \implies WP(s,P_2)$. Finally statement s induces a $must^-$-transition from b_1 to b_2 if $\forall V.P_2 \implies SP(s,P_1)$.

In order to show relations between the reachable states of M_A and M_C, it is useful to define a concretization function γ mapping states in S_A to states in S_C:

Definition 4.5: (*Concretization Function*). Let $\rho : S_C \times S_A$ be an abstraction relation. Let

$$\gamma_\rho(A) = \{s_c \mid \exists s_a \in A : (s_c, s_a) \in \rho\}$$

be the concretization function mapping a set of abstract states to its corresponding set of concrete states.

Using this function we define the set of initial abstract states I_A of M_A as the set satisfying the formula

$$I_C = \gamma_\rho(I_A)$$

When ρ is understood from context we will use γ rather than γ_ρ.

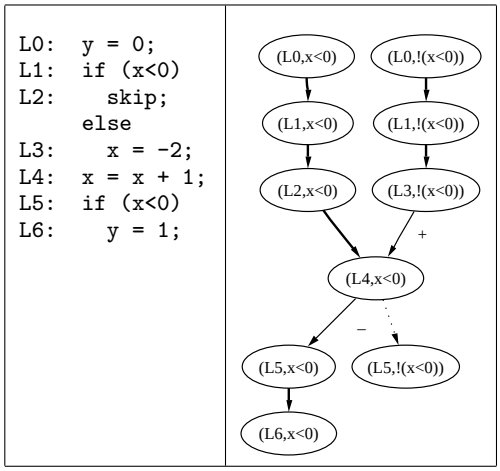

Figure 6: (a) The program from Figure 1(a) and (b) its abstract transitions.

4.4 Example

Figure 6(a) shows the program from Figure 1(a) and its set of (reachable) abstract transitions. Let us consider the statements in the program and the abstract transitions that they induce. The assignment statement at `L0` is "`y=0`". We have that $SP($`y=0,(x<0)`$) = WP($`y=0,(x<0)`$) =$ `(x<0)`. Therefore, we have a $must^\#$-transition `(L0,x<0)` $\xrightarrow{must^\#}$ `(L1,x<0)`. For similar reasons, we have the $must^\#$-transition `(L0,!(x<0))` $\xrightarrow{must^\#}$ `(L1,!(x<0))`.

The next statement is the `if`-statement at label `L1`. Because this statement branches exactly on the predicate `(x<0)`, it induces the $must^\#$-transitions:

$$\texttt{(L1,x<0)} \xrightarrow{must^\#} \texttt{(L2,x<0)}$$
$$\texttt{(L1,!(x<0))} \xrightarrow{must^\#} \texttt{(L3,!(x<0))}$$

The statement at label `L2` is a skip and so has no affect on the state, inducing the transition `(L2,x<0)` $\xrightarrow{must^\#}$ `(L4,x<0)`.

The assignment statement at label `L3` is reachable only when `!(x<0)` is true. It assigns the value -2 to variable `x`. We have that $WP($`x=-2,(x<0)`$) =$ `(-2<0)`, which reduces to true. This means that there is a $must^+$-transition `(L3,!(x<0))` $\xrightarrow{must^+}$ `(L4,(x<0)`. However, $WP($`x=-2,!(x<0)`$) =$ `(!(-2<0))`, which reduces to false. So there can be no transition from `(L3,!(x<0))` to (L4,`!(x<0))`. Now, let us consider strongest post-conditions. We have that $SP($`x=-2,!(x<0)`$) =$ `!(-2<0)`, which reduces to false, so there can be no $must^-$-transition from `(L3,!(x<0))` to `(L4,(x<0)`.

We now consider the assignment statement at label `L4` which is reachable only under `(x<0)` and which increments variable `x`. Because $SP($`x=x+1,(x<0)`$) =$ `(x<1)` and the set of states satisfying `(x<0)` is a subset of the set of states satisfying `(x<1)`, there is a $must^-$-transition `(L4,x<0)` $\xrightarrow{must^-}$ `(L5,x<0)`. There is no $must^+$-transition between these states because $WP($`x=x+1,(x<0)`$) =$ `(x<-1)` and the set of states satisfying `(x<0)` is not a subset of the set of states satisfying `(x<-1)`. The assignment statement induces a *may*-transition `(L4,x<0)` $\xrightarrow{may}$ `(L5,!(x<0))`. because this tran-

sition only takes places when variable `x` has the value -1 before the increment and the (resulting) value 0 after the increment.

Finally, there is a $must^{\#}$-transition `(L5,x<0)` $\overset{must^{\#}}{\longrightarrow}$ `(L6,x<0)` because the if-statement at label `L5` tests exactly the condition `(x<0)`.

5 Defining Predicate-Complete Test Coverage

Recall that the goal of predicate-complete testing (PCT) is to cover all reachable observable states, as defined by the m statements and n predicates $\Phi = \{\phi_1, \cdots, \phi_n\}$ in the program represented by the CTS M_C. The set of reachable observable states R is unknown, so we will use the Boolean (predicate) abstraction of M_C with respect to Φ to construct an abstract TTS M_A (see Definition 4.4).

In this section, we show how to analyze M_A to compute both upper and lower bounds to R. To do so, we find it useful to define a reachability function for a transition system. Let S be a set of states and δ be a transition relation of type $S \times S$. We define the reachability function over δ and $X \subseteq S$ as the least fixpoint of:

$$reach[\delta](X) = X \ \cup \ reach[\delta](\delta(X))$$

where $\delta(X)$ is the image of set X under δ.

We now define reachability in a CTS: Let M_C be a CTS. We denote the set of states reachable from states in T ($T \subseteq S_C$) as:

$$reach_C(T) = reach[\longrightarrow](T)$$

That is, reachability in M_C is simply defined as the transitive closure over the transitions in M_C, starting from states in T.

5.1 Upper Bound Computation

May-reachability in TTS M_A defines the upper bound U. Let M_C be a CTS and let M_A be an abstract TTS defined by abstraction relation ρ (via Definition 4.3). Consider $(s_c, s_a) \in \rho$. The upper bound is defined as:

$$U = reach[\overset{may}{\longrightarrow}_A](I_A)$$

That is, U is simply defined as the transitive closure over the *may*-transitions in M_A from the initial states I_A. It is easy to see that $reach_C(I_C) \subseteq \gamma(U)$, as the *may*-transitions of M_A overapproximate the set of transitions in M_C (by Definition 4.3). Since the reachable observable states R are contained in $reach_C(I_C)$, we have that U is an overapproximation of R.

5.2 Pessimistic Lower Bound Computation L_p

A set of abstract states $X \subseteq S_A$ is a lower bound of $R \subseteq S_C$ if for each $x_a \in X$, there is a $(x_c, x_a) \in \rho$ such that $x_c \in reach_C(I_C)$.

We present the computation of the lower bound in two steps. First, we define the pessimistic lower bound (L_p), which makes no assumptions about M_C. In the next section, we define the optimistic lower bound (L_o), which assumes that M_C does not diverge. We define L_p as:

$$\begin{aligned} L_p = \{\ v_a \ \mid \ & \exists t_a, u_a : \\ & t_a \in reach[\overset{must^-}{\longrightarrow}_A](I_A) \ \wedge \\ & (t_a \overset{may}{\longrightarrow}_A u_a \ \vee \ t_a = u_a) \ \wedge \\ & v_a \in reach[\overset{must^+}{\longrightarrow}_A](\{u_a\}) \ \} \end{aligned}$$

That is, an abstract state v_a is in L_p if there is a (possibly empty) sequence of $must^-$-transitions leading from $s_a \in I_A$ to t_a, there is a *may*-transition from t_a to u_a (or t_a is equal to u_a), and there is a (possibly empty) sequence of $must^+$-transitions from u_a to v_a.

We now show that for each $v_a \in L_p$, there is a $(v_c, v_a) \in \rho$ such that $v_c \in reach_C(I_C)$. That is, L_p is a lower bound to R. The proof is done in three steps, corresponding to the three parts of the definition of L_p:

- First, consider a sequence of $must^-$-transitions leading from $s_a \in I_A$ to t_a in M_A. Each $must^-$-transition $x_a \overset{must^-}{\longrightarrow}_A y_a$ identifies an *onto* relation from $\gamma(x_a)$ to $\gamma(y_a)$. That is, for all concrete states y_c mapping to y_a, there is a transition $x_c \longrightarrow y_c$ such that x_c maps to x_a. The transitive closure of an onto relation yields an onto relation. So, for all t_c mapping to t_a, we know that $t_c \in reach_C(I_C)$.
- Second, by the construction of M_A from M_C there is a *may*-transition $t_a \overset{may}{\longrightarrow}_A u_a$ only if there exists a transition $t_c \longrightarrow u_c$, where states t_c and u_c map to t_a and u_a, respectively. Since for all t_c mapping to t_a we know that $t_c \in reach_C(I_C)$, it follows that if there is a *may*-transition $t_a \overset{may}{\longrightarrow}_A u_a$ then there is some u_c mapping to u_a such that $u_c \in reach_C(I_C)$.
- Third, consider a sequence of $must^+$-transitions leading from t_a to v_a in M_A. Each $must^+$-transition $x_a \overset{must^+}{\longrightarrow}_A y_a$ identifies an *total* relation from $\gamma(x_a)$ to $\gamma(y_a)$. That is, for all concrete states x_c mapping to x_a, there is a transition $x_c \longrightarrow y_c$ such that y_c maps to y_a. The transitive closure of a total relation yields a total relation. So, for all t_c mapping to t_a, we know that there is a v_c mapping to v_a such that $v_c \in reach_C(\{t_c\})$.

This completes our proof that for each $v_a \in L_p$, there is a v_c mapping to v_a such that $v_c \in reach_C(I_C)$.

5.3 Optimistic Lower Bound Computation L_o

The optimistic computation of the lower bound (L_o) assumes that the program under consideration doesn't diverge (contain an infinite loop). We leave the problem of detecting non-termination to testing (which is typically done by defining some timeout threshold after which a computation is declared to be non-terminating).

We capture those states that must be reached (assuming convergence) using the idea of postdominance [All70]. In our context, a state t_a postdominates s_a in M_A if every path of *may*-transitions from s_a to a vertex in the set

$\{halt_A, error_A\}$ contains t_a. Postdominance in M_A is define as the greatest fixpoint to the following set of equations:

$$pd_A(halt_A) = \{halt_A\}$$

$$pd_A(error_A) = \{error_A\}$$

$$pd_A(s_a) = \{s_a\} \cup \bigcap_{s_a \xrightarrow{may}_A s'_a} pd_A(\{s'_a\})$$

Using L_p and pd_A, we define L_o as follows:

$$L_o = \{x_a \mid x_a \in pd_A(w_a) \wedge w_a \in L_p\}$$

It is easy to prove that if M_C contains no diverging computations then for all states w_c mapping to w_a, if $x_a \in pd_A(w_a)$ then there is a state x_c mapping to x_a such that $x_c \in reach_C(\{w_c\})$. Therefore, assuming that M_C contains no diverging computations, we have that for each $x_a \in L_o$, there is a x_c mapping to x_a such that $v_c \in reach_C(I_C)$.

We define the lower bound L as $L = L_o$.

5.4 Putting It All Together

Let M_A be the TTS constructed from concrete TTS M_C (via predicate abstraction on the set of predicates Φ, which induces an abstraction relation ρ). Let I_C be the set of initial concrete states of M_C and let I_A be the corresponding set of initial abstract states of M_A (induced by ρ).

In Figure 4, the ovals represent the initial abstract states

$$I_A = \{\ \texttt{L0}\text{:TTFT}, \texttt{L0}\text{:TTTT}, \texttt{L0}\text{:TTTF}, \texttt{L0}\text{:TTFF}\ \}$$

In Figure 4, all the nodes in the graph represent the set U while the rectangular nodes represent the set L and the plaintext nodes represent the set $U - L$. The shading of the rectangular nodes indicates the following:

- The white rectangular nodes represent those abstract states reachable from I_A via a sequence of $must^-$-transitions (in our example, these are $must^\#$-transitions which are, by definition, $must^-$-transitions). For example, consider the initial state L0:TTTF. There is a path of $must^\#$-transitions L0:TTTF $\xrightarrow{must^\#}_A$ L1:TTTF $\xrightarrow{must^\#}_A$ L2:TTTF $\xrightarrow{must^\#}_A$ L3:TTTF.
- The light-grey rectangular nodes (green in color) represent those abstract states only reachable via a sequence of $must^-$-transitions, followed by one *may*-transition, followed by a sequence of $must^+$-transitions. Thus, the set of ovals plus the set of white and light-grey rectangular nodes represents the set L_p. Consider the *may*-transition L3:TTTF $\xrightarrow{may}_A$ L4:FTTF, which continues the path given above. Covering this transition is the only way in which the state L4:FTTF can be reached. Then there is a path of $must^\#$-transitions (which, by definition, also are $must^+$-transitions): L4:FTTF $\xrightarrow{must^\#}_A$ L2:FTTF $\xrightarrow{must^\#}_A$ L3:FTTF. So, these three nodes are colored light-grey.
- Finally, the dark-grey rectangular nodes (red in color) represent those states in $L_o - L_p$. These are the states that must be reached under the assumption that the program does not diverge. In our example path, all the transitions leaving state L3:FTTF are *may*-transitions. Since any path to this state must contain a *may*-transition, the set L_p will not contain any of the *may*-successors of the state L3:FTTF. However, the state L4:FFFF is in L_o and so is colored dark-grey, as every path from L3:FTTF eventually leads to L4:FFFF (assuming loops terminate).

The path given above is one of the paths that leads to an array bounds error. Note that in this path the label `L3` is covered twice, once by the state L3:TTTF and then by the state L3:FTTF. In the first state, we have that `(lo<=hi)`, `(a[lo]<=pivot)` and `(a[hi]<=pivot)`. At label `L3`, `lo` is incremented by one. The path dictates (via the *may*-transition L3:TTTF $\xrightarrow{may}_A$ L4:FTTF) that the value of `lo` and `hi` are now equal. Because `(a[hi]<=pivot)` the loop at label `L2` continues to iterate and we reach the second state, L3:FTTF, in which we have that `(lo==hi)` and `(a[lo]<=pivot)` and `(a[hi]<=pivot)`. When `lo` is incremented the second time, its value becomes greater than `hi`, whose value still is the index of the last element of the array. Thus, the next access of `a[lo]` (see state L2:FFFF) will cause an array bounds violation.

6 Test Generation

The goal of test generation is to cover all the states in the lower bound L (plus any additional states, if we are lucky). Our test generation process consists of three steps:

- *Path Generation*: we use the set L_p to guide test generation. In particular, using this set, we identify a set of paths that are guaranteed to cover all states in L_o (if the program doesn't go wrong or enter an infinite loop).
- *Symbolic Execution*: we use symbolic execution on this set of paths in order to generate test data to cover these paths;
- *Observe Test Runs*: the program under test is run against this set of tests to check for errors and collect the set of executed observable states.

6.1 Path Generation

Let I_A be the set of initial abstract states in M_A. Consider the set of states L_p. The goal of the path generation phase is to enumerate all paths from I_A consisting of a sequence of $must^-$-transitions followed by one (and perhaps no) *may*-transition, while covering no state more than once. This can be done by a simple depth-first search procedure. The idea is that if we generate tests to cover these paths then we are guaranteed that the rest of the states in L_o will be covered if the execution of program does not go wrong (uncover an error) or diverge.

In Figure 4, using such a depth-first search identifies ten paths. These ten paths through L_p are uniquely identified by their beginning and ending vertices, as shown in the column "**Path Endpoints**" in Figure 7.

6.2 Symbolic Execution

Each of the ten paths induces a straight-line C "path" program that we automatically generated by tracing the path through the `partition` function. Consider the path from L0:TTTF to the L4:TTFF:

Path Endpoints	Generated Input Array	Bounds Error?
(L0:TTTT, L4:FTFT)	{ 0, -8, 1 }	no
(L0:TTTT, L4:TTFT)	{ 0, -8, 2, 1 }	no
(L0:TTTT, L4:TTTT)	{ 0, -8, -8, 1 }	no
(L0:TTTF, L4:TTFF)	{ 1, -7, 3, 0 }	no
(L0:TTTF, L4:FTTF)	{ 0, -7, -8 }	YES
(L0:TTTF, L4:TTTF)	{ 1, -7, -7, 0 }	YES
(L0:TTFT, L7:TTFF)	{ 0, 2, -8, 1 }	no
(L0:TTFT, L7:FTFT)	{ 0, 1, 2 }	no
(L0:TTFT, L7:TTFT)	{ 0, 3, 1, 2 }	no
(L0:TTFF, L0:TTTT)	{ 1, 2, -1, 0 }	no

Figure 7: The results of test generation for the running example.

```
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

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

Figure 8: The "path" program corresponding to the path L0:TTTF → L1:TTTF → L2:TTTF → L3:TTTF → L4:TTFF.

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

and its corresponding path program (see Figure 8). There are four 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 8. The four statements corresponding to the four transitions are presented after the "prelude" code in Figure 8. The `assert` statement at the end of the path program asserts that the final state at label L4 (TTFF) cannot occur, which of course is not true.

We used CBMC [CKY03], a bounded-model checker for C programs to generate a counterexample to the assertion that the state L4:TTFF cannot occur. CBMC produces an input array `a[]` and array length `n` that will cause the `assert` statement to fail, proving that L4:TTFF is reachable. For the generated path program of Figure 8, CBMC finds a counterexample and produces the input array { 1, -7, 3, 0 }, as shown in the second column of Figure 7.

6.3 Observe Test Runs

Instrumentation of the original program both collects the executed observable states for each test run and checks for array bounds violations. In our example, there are ten runs, two of which produce array bounds violations (because the `lo` index is incremented past the end of the input array and then `a[lo]` is accessed), as shown in the third column of Figure 7.

The set of observed states resulting from executing all ten tests contains all the states in Figure 4 except four of the states in $U - L$ (in particular, L5:FFFT, L6:FFFT and L7:FFFT and L3:FFTF) and the state L2:FFFF, which is unreachable due to an array bounds violation.

Fixing the error in the program and rerunning our entire process results in an upper bound U with 56 states and a lower bound L of 37 states. Test generation succeeds in covering all 37 states in the lower bound L and causes no array bounds errors. Additionally, these tests cover 6 of the 19 tests in $U - L$. This leads us to consider whether or not the remaining states in $U - L$ are reachable at all and to the problem of refining the upper and lower bounds.

7 Refinement of Lower and Upper Bounds

We now consider the problem of bring the lower bound L and upper bound U closer together. We focus our attention on the observable states in $U - L$ that were not covered by the test generation process of the previous section. The main question we wish to answer for these states is whether or not they are reachable in the original program. We can use automated machinery (such as the SLAM toolkit [BR02]) to try and answer this undecidable problem but, for many cases, will need to involve the programmer or tester.

Consider the state L7:FFFT from Figure 4, which is in $U - L$ and was not covered by any test. The concretization of this abstract state is

`lo>hi && a[lo]>pivot && a[hi]>pivot`

Notice that `partition` function, while having an array bounds error, does correctly maintain the invariant that all array elements with index less than the variable `lo` have value less than or equal to `pivot`. However, in the above state, we have that `hi<lo` and `a[hi]>pivot`. Thus, it is not possible to reach this state.

We submit that rather than ignore abstract states whose concrete counterparts are unreachable, it is important to introduce new predicates to try and eliminate such states in the abstraction. The reason is that these unreachable states often will point to boundary conditions that have not yet been tested. In order to eliminate the state L7:FFFT we will introduce three new predicates into the Boolean abstraction (in addition to the four already there) in order to track the status of the array when the variable `lo` takes on the value `hi+1`:

`(lo==hi+1)`, `(a[lo-1]<=pivot)`, `(a[hi+1]>pivot)`

These predicates track an important boundary condition that was not observed by the initial four predicates.

With these additional predicates, the generated Boolean abstraction has matching lower and upper bounds (actually $L_p = U$) and our test generation process covers all reachable observable states. As mentioned before, we can not expect to be able to achieve matching lower and upper bounds in general. Also, it is an open question how to generate such predicates automatically. Currently, the SLAM toolkit is not able to generate the above three predicates.

8 Discussion

We now consider what it means when set of states in the upper bound U and the pessimistic lower bound L_p are the same ($U = L_p$). We refer to this condition as "state simulation", as it means that every abstract state in the upper bound is observable by some execution of the concrete program.

We find it useful to informally describe the states represented by these two sets using regular expressions:

- $U = [I_A](\xrightarrow{may}_A)^*$;
- $L_p = [I_A](\xrightarrow{must^-}_A)^*(\xrightarrow{may}_A)^?(\xrightarrow{must^+}_A)^*$

That is, U is the set of abstract states reachable (from the initial set of abstract states I_A) via a sequence of *may*-transitions, while L_p is the set of states reachable from I_A via a sequence of $must^-$-transitions, followed by a most one *may*-transition, followed by a sequence of $must^+$-transitions.

Given a path p_a of abstract transitions in M_A containing more than one *may*-transition (not matched by a *must*-transition), it is impossible to know (without analysis of M_C) whether or not there exists a corresponding feasible execution path p_c in M_C. This is why the definition of L_p permits at most one *may*-transition (not matched by a *must*-transition).

State simulation is weaker than bisimulation [Mil99], which for deterministic systems (as we consider here) reduces to trace equivalence. An abstract TTS M_A bisimulates a CTS M_C if each *may*-transition in M_A is matched by a $must^+$-transition (i.e., $\xrightarrow{may}_A = \xrightarrow{must^+}_A$). It is easy to see that if M_A bisimulates M_C then every abstract state in U is reachable via a sequence of $must^+$-transitions. Therefore, $U = L_p$. Our use of $must^-$-transitions followed by a most one *may*-transition is the way in which we weaken bisimulation to state simulation.

Under state simulation, every abstract state s_a in U is reachable via a sequence of $must^-$-transitions, followed by a most one *may*-transition, followed by a sequence of $must^+$-transitions (which characterizes L_p). As we have shown previously, the existence of this sequence in M_A implies the existence of an execution trace in M_C in which s_a is observed. This implies that there is a finite set of tests sufficient to observe all states in U. Since U is an upper bound to the set of reachable observable states R this set of tests covers all states in R as well (that is, $R = U = L_p$).

The upper bound U is, by construction, sound (that is, $R \subseteq U$). State simulation implies that the least fixpoint of the *may*-abstraction induced by the set of observation predicates is complete (that is, $U \subseteq R$). [GRS00][6] In other words, the set U is equal to the set of observable states that would be encountered during the (infinite) computation of the least fixpoint over the concrete transition system M_C (with respect to the set of initial states I_C). This trivially follows from the fact that $L_p = U$. It follows that state simulation is a sufficient test for determining the completeness of a *may*-abstraction.

To summarize, the condition of state simulation ($U = L_p$) joins together the worlds of abstraction and testing. State simulation implies both a sound and complete abstract domain that can be completely covered by a finite set of tests.

[6] More precisely, *relatively* complete since we assume that abstract transitions can be computed precisely.

9 Related Work

Related work breaks into a number of topics.

9.1 Control-flow Coverage Criteria

We have already compared PCT coverage with statement, branch, multiple condition, predicate and path coverage (see Section 2). We now consider other alternatives to path coverage, namely linear code sequence and jump (LCSAJ) coverage and data-flow coverage based on def-use pairs. An LCSAJ represents a sequence of statements (which may contain conditional statements) ending with a branch. An LCSAJ is an acyclic path (no edge appears twice) through a control-flow graph ending with a branch statement. As we have shown, PCT coverage is incomparable to path coverage for loop-free programs, so it also is incomparable to LCSAJ coverage. The goal of def-use coverage is to cover, for each definition d of a variable x and subsequent use u of variable x, a path from d to u not containing another definition of x. If there is such a path from d to u then there is an acyclic path from d to u that doesn't contain another definition of x, so again PCT coverage is incomparable to def-use coverage.

9.2 Symbolic Execution and Test Generation

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 [BEL75, How76, Cla76, RHC] and continuing to the present day [JBW+94, GBR98, GMS98]. Recently, Chlipala et al. proposed using counterexample-driven refinement to guide test generation [CHJM04]. The major contribution of our work over previous efforts in this area is to guide test generation using Boolean abstraction and the computation of upper and lower bounds to the set of reachable observable states.

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 [BPS00]. Another way to limit the search is to bound the size of the input domain (say, to consider arrays of at most length three) [JV00], or to bound the maximum path length that will be considered, as done in bounded model checking [CKY03]. 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 [YM89]. 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 methods, our technique uses the set of input predicates to bound the set of paths that will be used to generate test data for a program. The predicates induce a Boolean abstraction that guides the selection of paths.

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 [Kor92]. The potential benefit of Korel's approach is that it is dynamic and has an accurate view of memory and flow depen-

dences. 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 [BKM02]. 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 may not work very well. Furthermore, it requires the user to supply a bound on the input size whereas our technique infers the input size.

Harder, Mellen and Ernst [HME03] propose using operational abstractions (properties inferred from observing a set of test executions) to guide the generation and maintenance of test suites. This is similar in spirit to predicate-complete testing but unsound (the properties inferred are "likely" invariants but not guaranteed to hold in general). In contrast, our use of predicate abstraction and reachability analysis in the abstract domain computes a (sound) overapproximation to the set of reachable observable states of a program. Furthermore, the invariants we can establish about a program's behavior involve arbitrary boolean expressions over atomic predicates whereas Harder et al. limit themselves to atomic predicates and implications between atomic predicates.

9.3 Three-valued Model Checking

Our work was inspired by the work on three-valued model checking by Bruns, Godefroid, Huth and Jagadeesan [BG99, GHJ01, GR03]. Their work shows how to model incomplete (abstract) systems using modal transition systems (equivalently, partial Kripke Structures), as we have done here. It then gives algorithms for model checking temporal logic formula with respect to such systems. Given an MTS, these algorithms can determine whether a temporal logic formula is definitely true, definitely false or unknown with respect to the MTS.

Our computation of lower and upper bounds achieves a similar result but infers reachability properties of a concrete TTS M_C from analysis of an abstract TTS M_A. The lower bound L characterizes those observable states that are definitely reachable, the upper bound U (more precisely, its inverse $S-U$) characterizes those observable states that are definitely not reachable, and the reachability status of states in $U-L$ are unknown.

To achieve a precise lower bound for reachability, we generalized the definition of *must*-transitions given for MTS to account for three types of *must*-transitions: $must^+$ (which correspond to *must*-transitions in an MTS), $must^-$ and $must^\#$.

In model checking of abstractions of concrete transition systems, one is interested in proving that a temporal property holds for *all* concrete execution paths starting from some initial abstract state. This is the reason why only $must^+$-transitions are used in model checking of modal transitions systems. For reachability, one is interested proving the existence of *some* concrete execution path starting from some initial abstract state. Thus, $must^-$-transitions are of interest.

10 Conclusion

We have presented a new form of control-flow coverage that is based on observing the vector consisting of a program's conditional predicates, thus creating a finite-state space. There are a number of open questions to consider. First, what is a logical characterization of tri-modal transition systems? Second, how can one automate the refinement process to bring the lower and upper bounds closer? (It is well known that the set of *must*-transitions is not generally monotonically non-decreasing when predicates are added to refine an abstract system. Recently, Shoham and Grumberg [SG04] and Alfaro, Godefroid and Jagadeesan [dAGJ04] independently proposed a new form of *must*-transition that permits monotonic refinement of abstractions.) Finally, how does this technique work in practice?

Acknowledgements

Thanks to Daniel Kroening for his help with the CBMC model checker. Thanks also to Byron Cook, Tony Hoare, Vladimir Levin, Orna Kupferman and Andreas Podelski for their comments.

References

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.

[BEL75] 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.

[BG99] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *CAV 99: Computer Aided Verification*, LNCS 1633, pages 274–287. Springer-Verlag, 1999.

[BKM02] 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.

[BMMR01] 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.

[BPS00] 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.

[BR00] 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.

[BR02] 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.

[CHJM04] A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE 04: International Conference on Software Engineering (to appear)*. ACM, 2004.

[CKY03] 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.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[dAGJ04] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games:uncertainty, but with precision. In *LICS 04: Logic in Computer Science*, To appear in LNCS. Springer-Verlag, 2004.

[GBR98] 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.

[GHJ01] P. Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR 01: Conference on Concurrency Theory*, LNCS 2154, pages 426–440. Springer-Verlag, 2001.

[GMS98] 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.

[God03] P. Godefroid. Reasoning about abstract open systems with generalized module checking. In *EMSOFT 03: Conference on Embedded Software*, LNCS 2855, pages 223–240. Springer-Verlag, 2003.

[GR03] P. Godefroid and R.Jagadeesan. On the expressiveness of 3-valued models. In *VMCAI 03: Verification, Model Checking and Abstract Interpretation*, LNCS 2575, pages 206–222. Springer-Verlag, 2003.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[HME03] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE 2003: International Conference on Software Engineering*, pages 60–71. ACM, 2003.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2:208–215, 1976.

[JBW+94] 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.

[JV00] 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.

[Kor92] B. Korel. Dynamic method of software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

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

[SG04] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for ctl. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2988, pages 546–560. Springer-Verlag, 2004.

[Tai96] K-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, 1996.

[Tai03] K-C. Tai. Predicate-based test generation for computer programs. In *ICSE 03: International Conference on Software Engineering*, pages 267–276, 2003.

[YM89] 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.