On the Limit of Control Flow Analysis for Regression Test Selection

Thomas Ball
Lucent Technologies, Bell Laboratories
1000 E. Warrenville Rd., Room 1G-359
Naperville, IL 60566 USA
tball@research.bell-labs.com

Abstract

Automated analyses for regression test selection (RTS) attempt to determine if a modified program, when run on a test t, will have the same behavior as an old version of the program run on t, but without running the new program on t. RTS analyses must confront a price/performance tradeoff: a more precise analysis might be able to eliminate more tests, but could take much longer to run.

We focus on the application of control flow analysis and control flow coverage, relatively inexpensive analyses, to the RTS problem, considering how the precision of RTS algorithms can be affected by the type of coverage information collected. We define a strong optimality condition (edge-optimality) for RTS algorithms based on edge coverage that precisely captures when such an algorithm will report that re-testing is needed, when, in actuality, it is not. We reformulate Rothermel and Harrold's RTS algorithm and present three new algorithms that improve on it, culminating in an edge-optimal algorithm. Finally, we consider how path coverage can be used to improve the precision of RTS algorithms.

[1]

Keywords

Regression testing, control flow analysis, coverage, profiling

1 Introduction

The goal of regression test selection (RTS) analysis is to answer the following question as inexpensively as possible:

> Given test input t and programs *old* and *new*, does $new(t)$ have the same observable behavior as $old(t)$?

Of course, it is desired to answer this question *without* running program *new* on test t. RTS analysis uses static analysis of programs *old* and *new* in combination with dynamic information (such as coverage information) collected about the execution $old(t)$ in order to make this determination. An RTS algorithm either *selects* a test for re-testing or *eliminates* the test.

Static analyses for RTS come in many varieties: some examine the syntactic structure of a program [6]; others use control flow or control dependence information [11, 12]; more ambitious analyses examine the def-use chains or flow dependences of a program [9, 5]. Typically, each of these analyses is more precise than the previous, but at a greater cost.

A safe (conservative) RTS analysis never eliminates a test t if $new(t)$ has different behavior than $old(t)$. A safe algorithm may select some test when it could have been eliminated.

We focus on the application of control flow analysis to safe regression testing (from now on we will use CRTS to refer to "Control-flow-based RTS"). Previous work has improved the precision of CRTS analysis but left open the question of what the limit of such analyses are. CRTS can be improved in two ways: by increasing the precision of the analysis applied to the control flow graph representations of programs *old* and *new*, or by increasing the precision of the dynamic information recorded about the execution $old(t)$. We will address both issues and the interactions between them.

Our results are threefold:

- (Section 3) Building on recent work in CRTS by Rothermel and Harrold [12], we show a strong relationship between CRTS, deterministic finite state automata, and the intersection of regular languages. We define the *intersection graph* of two control flow graphs, which precisely captures the goal of CRTS and forms the basis for a family of CRTS algorithms, parameterized by what dynamic information is collected about $old(t)$.
- (Section 4) We consider the power of CRTS when the dynamic information recorded about $old(t)$ is edge coverage (i.e., whether or not each edge of *old*'s control flow graph was executed). We define a strong optimality condition (edge-optimality) for CRTS algorithms based on edge coverage. We then reformulate Rothermel and Harrold's CRTS algorithm in terms of the intersection graph and present three new algorithms that improve on it, culminating in an edge-optimal algorithm. The first algorithm eliminates a test whenever the Rothermel/Harrold algorithm does, and safely eliminates more tests in general, at the same cost. The next two algorithms are even more precise, but at greater computational cost.

[1] To appear, 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis

- (Section 5) By recording path coverage information about $old(t)$ rather than edge coverage, we can improve upon edge-optimal algorithms. However, if path profiling is limited to tracking paths of a bounded length (which is motivated by concerns of efficiency), then an adversary will always be able to choose a program *new* that will cause any CRTS algorithm based on path coverage to fail.

Section 6 reviews related work and Section 7 summarizes the paper.

2 Background

We assume a standard imperative language such as C, C++, or Java in which the control flow graph of a procedure P is completely determined at compile time. In P's control flow graph G, each vertex represents a basic block of instructions and each edge represents a control transition between blocks. The translation of an abstract syntax tree representation of a procedure into its control flow graph representation is well known [1]. Since G is an executable representation of P, we will talk about executing both P and G on a test t.

We now define some graph terminology that will be useful in the sequel.

Let $G = (V,\ E,\ s,\ x)$ be a directed control flow graph with vertices V, edges E, a unique entry vertex s, from which all vertices are reachable, and exit vertex x, which has no successors and is reachable from all vertices. A vertex v is labelled with $BB(v)$, the code of the basic block it contains. Two different vertices may have identical labels. It is often convenient to refer to a vertex by its label and we will often do so, distinguishing vertices with identical labels when necessary.

An edge $e = v \rightarrow_l w$ connects source vertex v to target vertex w via a directed edge labelled l. The outgoing edges of each vertex are uniquely labeled. Labels are values (typically, *true* or *false* for boolean predicates) that determine where control will transfer next after execution of $BB(v)$.[2] If a vertex v has only one outgoing edge, its label is ε, which is not shown in the figures.

Since the outgoing edges of a vertex are uniquely labelled, an edge $v \rightarrow_l w$ may also be represented by a pair (v, l), which we call a *control transition* or transition, for short. The vertex $succ(v, l)$ denotes the vertex that is the l-successor of vertex v (if $v \rightarrow_l w$ then $w = succ(v, l)$).

A path in G is a sequence of edges

$$p = [e_1, e_2, \ldots, e_n]$$

where the target vertex of e_i is the source vertex of e_{i+1} for $1 \leq i < n$. A path may be represented equivalently by an alternating sequence of vertices and edge labels

$$p = [v_1, l_1, v_2, l_2, \ldots, l_n, v_{n+1}]$$

where v_i is the source vertex of edge e_i (for $1 \leq i \leq n$), v_{n+1} is the target vertex of e_n, and l_i is the label of edge e_i (for $1 \leq i \leq n$). Given a path p of n edges (and $n+1$ vertices), let $p_v[i]$ be the ith vertex ($1 \leq i \leq n+1$), and let $p_l[i]$ be the ith edge label ($1 \leq i \leq n$).

[2]The number of outgoing edges of vertex v and the labels on these edges are uniquely defined by $BB(v)$. Thus, different vertices that have identical basic blocks will have the same number of outgoing edges with identical labels.

Paths beginning at a designated vertex (for our purposes, the entry vertex s) are equivalently represented by a sequence of basic blocks and labels (rather than a sequence of edges or vertices and labels):

$$p = [BB(v_1), l_1, BB(v_2), l_2, \ldots, l_n, BB(v_{n+1})]$$

A *complete path* is a path from s to x.

Figure 1 shows two programs P and P' and their corresponding control flow graphs G and G'. For both G and G', the entry vertex is $s = A$ and the exit vertex is $x = X$. The label of a vertex v denotes its basic block $BB(v)$. Graph G has one occurrence of basic block C while graph G' has two occurrences of C. The graph G'' is the intersection graph of G and G' and is discussed next.

3 CRTS and the Intersection Graph

In control flow analysis, the graphical structure of a program is analyzed, but the semantics of the statements in a program are not, except to say whether or not two statements are textually identical. This implies that CRTS algorithms must assume that every complete path through a graph is potentially executable (even though there may be unexecutable paths). Unexecutable paths cannot affect the safety of CRTS algorithms, but may decrease their precision, just as they do in compiler optimization.

CRTS algorithms must be able to determine if two basic blocks are semantically equivalent. Of course, this is undecidable in general. Following Rothermel and Harrold, we use textual equivalence of the code as a conservative approximation to semantic equivalence, which is captured in the definition of *equivalent* vertices: Two vertices v and w (from potentially different graphs) are *equivalent* if the code of $BB(v)$ is lexicographically identical to $BB(w)$. Let $Equiv(v, w)$ be true iff v is equivalent to w.[3]

Once we have equivalent vertices, we can extend equivalence to paths, as follows: paths p and q are *identical* if p and q are the same length, $Equiv(p_v[i], q_v[i])$ is true for all i, and $p_l[i] = q_l[i]$ for all i. That is, p and q are identical words (over an alphabet of basic blocks and labels).

The following simple definition (a restatement of that found in [11]) precisely captures the power of CRTS:

> If graph G run on input t (denoted by $G(t)$) traverses complete path p and graph G' contains complete path p' identical to p, then $G'(t)$ will traverse path p' and have the same observable behavior as $G(t)$.

The above definition translates trivially into the most precise and computationally expensive CRTS algorithm: record the complete execution path of $G(t)$ (via code instrumentation that traces the path [4]) and compare it to the control flow graph of G' to determine if the path exists there. We will see later in Section 5 that any algorithm that does not record the complete execution path of $G(t)$ can be forced, by an adversary choosing an appropriate graph G', to select a test that could have been eliminated.

We observe that a control flow graph G may be viewed as a deterministic finite automaton (DFA) with start state s and final state x that accepts the language $L(G)$, the set of all complete paths in G. More precisely, a control flow graph G has a straightforward interpretation as a DFA in which each vertex v in V corresponds to two

[3]The exit vertex x can only be equivalent to other exit vertices (i.e., vertices with no successors).

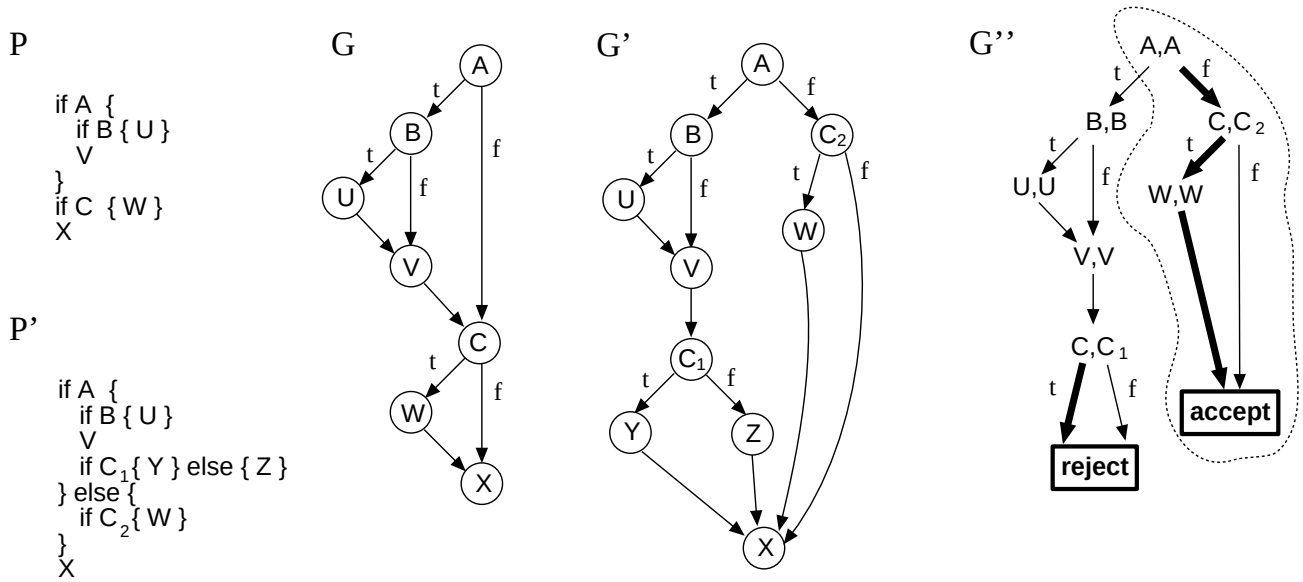

Figure 1: Example programs P and P', their corresponding control flow graphs G and G', and the intersection graph G'' of G and G'.

states, v_1 and v_2. These states are connected by a state transition $v_1 \rightarrow_{BB(v)} v_2$, labelled by $BB(v)$. Edges in E are also interpreted as state transitions: an edge $v \rightarrow_l w$ is interpreted as a state transition $v_2 \rightarrow_l w_1$. The alphabet of the DFA is the union of all basic blocks and all edge labels, s_1 is the start state, and x_2 is the final state. The DFA recognizes precisely the complete paths of G. Rather than represent the control flow graph in this more verbose fashion, we choose to present it in its traditional form but keep its DFA interpretation in mind.

Given this insight, the CRTS question reduces to:

Is a complete path p from $L(G)$ also in $L(G')$?

Let $I(G,G') = L(G) \cap L(G')$, the paths for which re-testing is not needed, and let $D(G,G') = L(G) - L(G')$, the paths for which re-testing is needed.

A CRTS algorithm is *optimal* if, given any path p in $I(G,G')$, the algorithm reports that p is in $I(G,G')$. A CRTS algorithm is *safe* if, given any path p in $D(G,G')$, the algorithm reports that p is in $D(G,G')$.

To help reason about $I(G,G')$ and $D(G,G')$, we define a new graph $G'' = (V'', E'')$, the intersection graph of $G = (V,E,s,x)$ and $G' = (V',E',s',x')$, which also has a straightforward interpretation as a DFA.[4] This graph can be efficiently constructed from G and G'.

The vertex set V'' of G'' is simply the cross product of V and V', with two additional vertices:

$$V'' = (V \times V') \cup \{accept, reject\}$$

We use the following relation to help define E'':

$$(v,v') \hookrightarrow_l (w,w') = \\ Equiv(v,v'), v \rightarrow_l w \in E \text{ and } v' \rightarrow_l w' \in E'$$

[4] G'' is essentially an optimized version of a product automaton of G and G' [7].

The edge set E'' is defined in terms of $\hookrightarrow_l$ and the $Equiv$ relation.

$$\begin{aligned} E'' \;=\; & \{ \; (v,v') \rightarrow_l (w,w') \mid (v,v') \hookrightarrow_l (w,w'), \\ & \quad Equiv(w,w'), w \neq x \; \} \\ \cup \; & \{ \; (v,v') \rightarrow_l accept \mid (v,v') \hookrightarrow_l (x,x'), \\ & \quad Equiv(x,x') \; \} \\ \cup \; & \{ \; (v,v') \rightarrow_l reject \mid (v,v') \hookrightarrow_l (w,w'), \\ & \quad \text{not } Equiv(w,w') \; \} \end{aligned}$$

Vertex (s,s') is the entry vertex of G''. We will restrict the vertex and edge sets of G'' to be the vertices and edges reachable from (s,s'). If no vertices (other than (s,s')) or edges are reachable from (s,s') then s and s' are not equivalent. A pair (v,v') is reachable from (s,s') iff there is a path p in G from s to v that is a prefix of a path in $I(G,G')$. Vertex *reject* represents the reject state, which corresponds to paths in $D(G,G')$. Vertex *accept* represents the accept state, which corresponds to paths in $I(G,G')$.

Figure 1 shows the intersection graph G'' of the graphs G and G' in the figure. We can see that there are two paths in $I(G,G')$, corresponding to the paths:

$$[(A,A), f, (C,C_2), f, accept]$$

and

$$[(A,A), f, (C,C_2), t, (W,W), \varepsilon, accept]$$

in G''. The corresponding paths in G are: $[A, f, C, f, X]$ and $[A, f, C, t, W, \varepsilon, X]$. Graph G'' also shows that any path that begins with the transition (A,t) is in $D(G,G')$.

Two straightforward results about the intersection graph G'' will inform the rest of the paper: A path p is in $I(G,G')$ iff it is represented by a path from (s,s') to *accept* in G''; a path p is in $D(G,G')$ iff it is represented by a path from (s,s') to *reject* in G''. Of course, every complete path p in G is either in $I(G,G')$ or $D(G,G')$. More formally:

Theorem 1 *Let G'' be the intersection graph of graphs G and G'. Path*

$$p = [s,\ l_1,\ v_2,\ l_2,\ \ldots,\ l_n,\ x]$$

from G is in $I(G,G')$ iff

$$[(s,s'),\ l_1,\ (v_2,v_2'),\ l_2,\ \ldots,\ l_n,\ accept]$$

is in G''.

Theorem 2 *Let G'' be the intersection graph of graphs G and G'. Path*

$$p = [s = v_1,\ l_1,\ v_2,\ l_2,\ \ldots,\ l_n,\ x = v_{n+1}]$$

from G is in $D(G,G')$ iff there exists i $(1 \le i \le n)$ such that

$$[(v_1,v_1'),\ l_1,\ (v_2,v_2'),\ l_2,\ \ldots,\ l_i,\ reject]$$

is in G''.

Figure 2 shows how the intersection graph of graphs G and G' is computed via a synchronous depth-first search of both graphs. The procedure DFS is always called with equivalent vertices v and v'. If (v,v') is already in V'', this pair has been visited before and the procedure returns. Otherwise, (v,v') is inserted into V'' and each edge $v \rightarrow_l w$ and its corresponding edge $v' \rightarrow_l w'$ is considered in turn.[5] Edges are appropriately inserted into E'' to reflect whether or not vertices w and w' are equivalent, and whether or not w is the exit vertex of G. The algorithm recurses only when w and w' are equivalent and w is not the exit vertex of G. The algorithm also computes the set of vertices V''_{accept} from which *accept* is reachable in G'', which will be used later.

The worst-case time complexity of the algorithm is $O(|E| \cdot |E'|)$. Note that it is not necessary to store the relation E'' explicitly, since it can be derived on demand from V'', E and E'. Thus, the space complexity for storing the intersection graph (as well as V''_{accept}) is $O(|V| \cdot |V'|)$ in the worst case.

4 CRTS Using Edge Coverage

What is the limit of CRTS given that the dynamic information collected about $G(t)$ is edge coverage? Consider a complete path p representing the execution path of $G(t)$ and the set of edges E_p of G that it covers. There may be another complete path q in G, distinct from p, such that $E_q = E_p$.[6] Let P_p represent the set of paths (including p) whose edge sets are identical to E_p.

To determine whether or not G' needs retesting, a CRTS algorithm using edge coverage must consider (at least implicitly) all the paths in P_p. If all of these paths are members of $I(G,G')$ then the CRTS algorithm can and should eliminate the test that generated path p. However, if even one of the paths in P_p is in $D(G,G')$ then the algorithm must select the test in order to be safe.

Given this insight, we can now define what it means for a CRTS algorithm to be *edge-optimal*:

> A CRTS algorithm is *edge-optimal* if for any path p such that $P_p \subseteq I(G,G')$, the algorithm reports that p is in $I(G,G')$.

[5]Note that if v and v' are equivalent then $w' = succ(v',l)$ must be defined since $BB(v')$ is identical to $BB(v)$.

[6]Note that no such paths can exist if G is acyclic. In this case, each complete path has a different set of edges than all other complete paths.

```
V'' := {reject, accept}
V''_accept := {accept}
if Equiv(s, s') then
  DFS (s, s')
fi

procedure DFS(v, v')
begin
  if (v, v') ∉ V'' then
    V'' := V'' ∪ {(v, v')}
    for each edge v →_l w ∈ E(G) do
        let w' = succ(v', l) in
            if Equiv(w, w') then
              if w = x then
                E'' := E'' ∪ {(v, v') →_l accept}
              else
                E'' := E'' ∪ {(v, v') →_l (w, w')}
                DFS(w, w')
              fi
            else
              E'' := E'' ∪ {(v, v') →_l reject}
            fi
            if (w, w') ∈ V''_accept then
              V''_accept := V''_accept ∪ {(v, v')}
            fi
        ni
    od
  fi
end
```

Figure 2: Constructing the intersection graph of G and G' via a synchronous depth-first search of the two graphs. The algorithm also determines the set of vertices V''_{accept} from which *accept* is reachable.

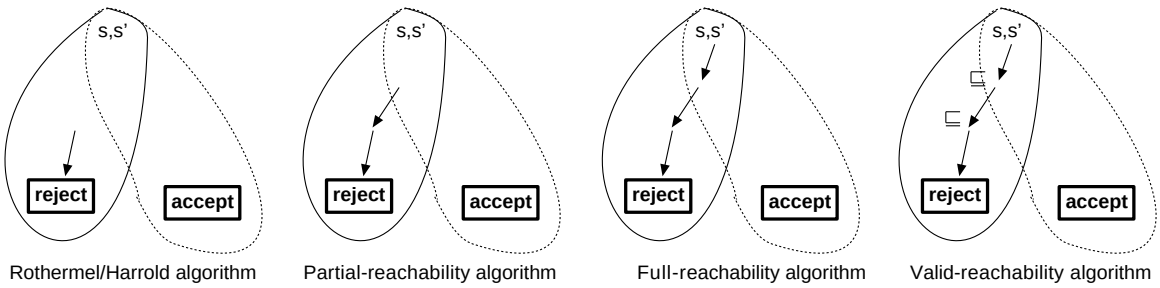

Figure 3: The four edge-based CRTS algorithms, summarized pictorially with the intersection graph. The dotted outline represents V''_{accept}, the vertices of G'' from which *accept* is reachable.

Algorithm	Time	Space	Precision	Edge-optimal?
Rothermel/Harrold	$O(\lvert E\rvert \cdot (\lvert E'\rvert + \lvert T\rvert))$	$O(\lvert V\rvert \cdot \lvert V'\rvert)$	$>$	no
Partial-reachability	$O(\lvert E\rvert \cdot (\lvert E'\rvert + \lvert T\rvert))$	$O(\lvert V\rvert \cdot \lvert V'\rvert)$	$>>$	no
Full-reachability	$O(\lvert E\rvert \cdot \lvert E'\rvert \cdot \lvert T\rvert)$	$O(\lvert V\rvert \cdot \lvert V'\rvert)$	$>>>$	no
Valid-reachability	$O(\lvert E\rvert \cdot \lvert E'\rvert \cdot \lvert T\rvert)$	$O(\lvert V\rvert \cdot \lvert V'\rvert)$	$>>>>$	yes

Table 1: Comparison of four edge-based CRTS algorithms.

We first present the Rothermel/Harrold (RH) algorithm, restated in terms of the intersection graph. We then present three new algorithms, culminating in an edge-optimal algorithm. Figure 3 illustrates what the RH algorithm and each of the four algorithms does, using the intersection graph.[7] Each picture shows the start vertex (s,s'), and final states *reject* and *accept*. The dotted outline represents V''_{accept}, the vertices of G'' from which *accept* is reachable.

- The RH algorithm detects whether or not E_p covers an edge incident to *reject*. If it does not, then path p must be in $I(G,G')$.
- The partial-reachability algorithm detects whether or not E_p covers a path in the intersection graph from an edge leaving V''_{accept} to the *reject* vertex. Again, if no such path exists then p is in $I(G,G')$. A surprising result is that partial-reachability of *reject* can be determined with time and space complexity equivalent to the RH algorithm. This algorithm is more precise than the RH algorithm since it may be the case that E_p contains an edge incident to *reject* but does not cover a partial path from a vertex in V''_{accept} to *reject*.
- The full-reachability algorithm determines whether or not E_p covers a path from (s,s') to *reject*. If not, then p is in $I(G,G')$. This algorithm is more precise than the partial-reachability algorithm, but at a greater cost. However, it is still not edge-optimal.
- The valid-reachability algorithm makes use of a partial order $\sqsubseteq$ on edges in G to rule out certain "invalid" paths. We show that if $P_p \subseteq I(G,G')$ then E_p cannot cover a valid reaching path to *reject* from (s,s'), yielding an edge-optimal algorithm.

Table 1 summarizes the time and space complexity for the four algorithms. T represents the set of tests on which G has been run. All edge-based CRTS algorithms incur a storage cost of $O(|E|\cdot|T|)$ for the edge coverage information stored for each test in T, which we factor out when discussing the space complexity of these algorithms.

[7] If s is not equivalent to s', then $I(G,G')$ is empty. We assume that all four algorithms initially check this simple condition before proceeding.

4.1 The Rothermel-Harrold Algorithm

We now present the RH algorithm in terms of the intersection graph $G'' = (V'', E'')$. The RH algorithm first computes the set D of control transitions incident to *reject* (using a synchronous depth-first search of graphs G and G' similar to that in Figure 2):

$$D = \{(v,l) \mid ((v,v') \rightarrow_l \textit{reject}) \in E''\}$$

Given D and an edge set E_p, the RH algorithm then operates as follows: If $E_p \cap D = \emptyset$ then path p must be in $I(G,G')$ since it contains no transition from D, which is required for p to be in $D(G,G')$. Otherwise, conservatively assume that p is in $D(G,G')$.

Consider the intersection graph of Figure 1. For this graph, $D = \{(C,t),(C,f)\}$. Since every path from A to X in graph G contains one of these transitions, the RH algorithm will require all tests to be rerun on G'. However, in this example, for any path p in $I(G,G')$, $P_p \subseteq I(G,G')$, so the RH algorithm is not edge-optimal. Consider such a path

$$p = [A, f, C, t, W, \varepsilon, X]$$

The transitions of G'' covered by E_p are shown as bold edges in Figure 1. There is no complete path other than p that covers exactly the transitions (A,f), (C,t) and (W,ε).

The time and space complexity to compute D is clearly the same as that for the depth-first search algorithm of Figure 2. To compute, for all tests t in a set of tests T, whether or not the set of edges covered by $G(t)$ contains a transition from D, takes $O(|E|\cdot|T|)$ time. Thus, the RH algorithm has an overall running time of $O(|E|\cdot(|E'|+|T|))$ and space complexity of $O(|V|\cdot|V'|)$.

Rothermel and Harrold show that if G and G' do not have a "multiply-visited vertex" then their algorithm will never report that p is in $D(G,G')$ when it actually is in $I(G,G')$. This means that their algorithm is optimal (and thus edge-optimal) for this class of graphs. Stated in terms of the intersection graph G'', a vertex v in G is a "multiply-visited vertex" if:

$$|\{(v,v') \in V''\}| > 1$$

So in Figure 1, vertex C of graph G is a multiply-visited vertex. Rothermel and Harrold ran their algorithm on a set of seven small

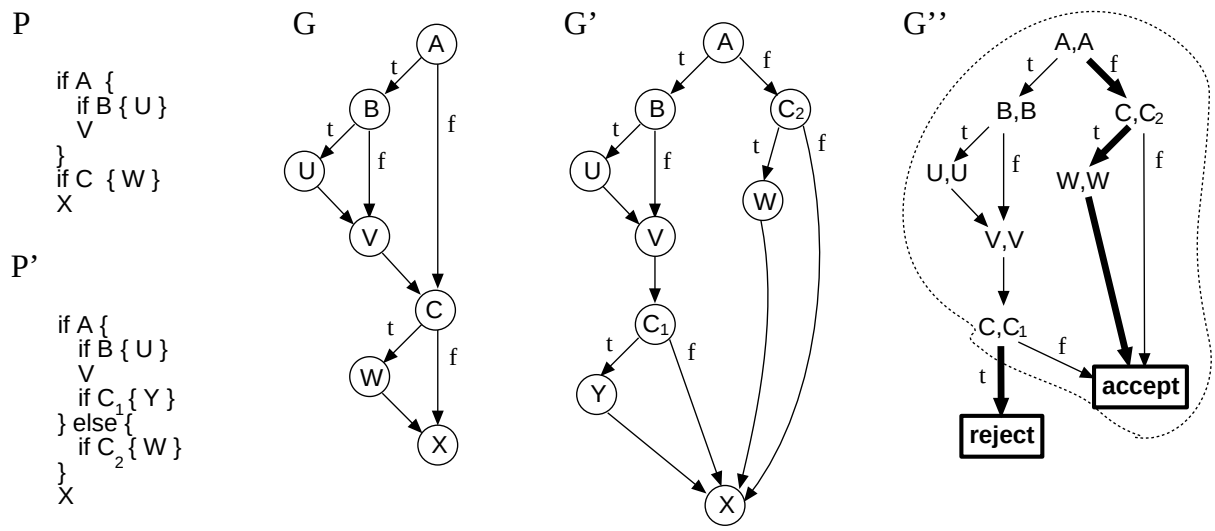

Figure 4: An example that shows that the partial-reachability algorithm is not edge-optimal.

programs (141-512 lines of code, 132 modified versions) and one larger program (49,000 lines of code, 5 modified versions), and found that the multiply-visited vertex condition did not occur for these programs and their versions [12]. Further experimentation is clearly needed on larger and more diverse sets of programs to see how often this condition arises.

4.2 The Partial-reachability Algorithm

Let us reconsider the example of Figure 1. The dotted outline in graph G'' shows the set V''_{accept}. The only transition leaving this set is (A,t). Any path leading to *reject* must include this transition. Thus, if this transition is not in E_p then p must be in $I(G,G')$, as is the case with path

$$p = [A, f, C, t, W, \varepsilon, X]$$

which has $E_p = \{(A,f),(C,t),(W,\varepsilon)\}$.

Consider the projection of E_p onto the edge set of G'':

$$E''_p = \{((v,v'),l) \in E'' \mid (v,l) \in E_p\}$$

and the graph $G''_p = (V'', E''_p)$ that results (the edges of E''_p are shown in bold in Figure 1). It is straightforward to see that, in general, for any edge $v'' \to w''$ in G''_p, either *accept* or *reject* must be reachable from w'' in G''_p. Therefore, for an edge $v'' \to w''$ in G''_p, if v'' is in V''_{accept} and w'' is not in V''_{accept} it must be the case that *reject* is reachable from w''.

This observation leads to the partial-reachability algorithm which has time and space complexity identical to that of the RH algorithm, yet is more precise. This algorithm does not require construction of G''_p, but is able to determine whether or not *reject* is partially-reachable from an edge leaving V''_{accept}.

Similar to the RH algorithm, this algorithm first computes a set D_{reject} of transitions in G using the intersection graph:

$$D_{reject} = \{\ (v,l)\ \mid\ ((v,v') \to_l w'') \in E'',\ (v,v') \in V''_{accept}, w'' \notin V''_{accept}\}$$

The set D_{reject} contains transitions in G that transfer control out of V''_{accept}. The algorithm then operates as follows: If $E_p \cap D_{reject} = \emptyset$ then p is in $I(G,G')$, since p must contain a transition from D_{reject} in order to be in $D(G,G')$. Otherwise, conservatively assume that p is in $D(G,G')$.

It is easy to see that the partial-reachability algorithm subsumes the RH algorithm, since whenever $E_p \cap D_{reject}$ is not empty, $E_p \cap D$ also will not be empty. Stated another way, whenever the RH algorithm reports that p is in $I(G,G')$, the partial-reachability algorithm will report the same.

As shown in Figure 2, the set V''_{accept} can be determined during construction of the intersection graph, in $O(|E| \cdot |E'|)$ time and $O(|V| \cdot |V'|)$ space. To compute D_{reject} takes $O(|E| \cdot |E'|)$ time and simply requires visiting every edge e'' in E'' to determine if e'' leaves V''_{accept}. If so, then the transition e in G corresponding to e'' is added to D_{reject}. Once D_{reject} has been computed, the rest of the algorithm is identical to the RH algorithm: for each test in T, check whether or not the set of edges covered by the test has an edge in D_{reject}. Thus, the time and space complexity of this algorithm is identical to the RH algorithm.

4.3 The Full-reachability Algorithm

Figure 4 shows that the partial-reachability algorithm is not edge-optimal. In this example, the intersection graph G'' has

$$D = D_{reject} = \{(C,t)\}$$

Thus, for the path $p = [A, f, C, t, W, \varepsilon, X]$, which is in $I(G,G')$ and for which $P_p \subseteq I(G,G')$, both the RH algorithm and partial-reachability algorithm will fail to report that p is in $I(G,G')$ since transition (C,t) is covered by path p. Note, however, that in G''_p the *reject* vertex is not reachable from (s,s').

In general, either *reject* or *accept* must be reachable from (s,s') in G''_p. The full-reachability algorithm is simple: If *reject* is not reachable in G''_p then p is in $I(G,G')$. Otherwise, conservatively assume that p is in $D(G,G')$.

Consider graph G in Figure 4. Any complete path in G containing

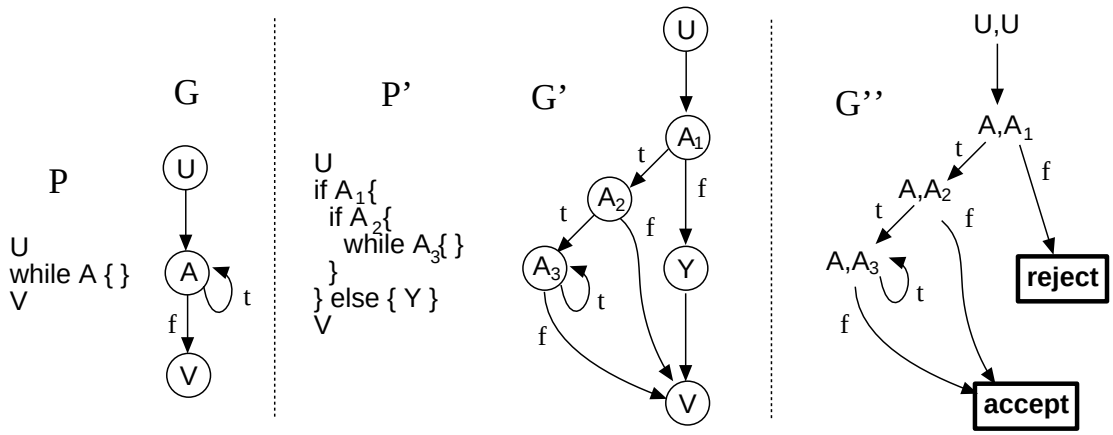

Figure 5: An example that shows that the full-reachability algorithm is not edge-optimal.

the transition (A, f) is in $I(G, G')$ and, additionally, does not contain the transition (A, t). Therefore, for any such path p, vertex *reject* is not reachable from vertex (A, A) in G''_p.

The DFS algorithm in Figure 2 can be easily modified to compute the reachability of *reject* in G''_p, but must be run for each test in T, resulting in an overall running time of $O(|E| \cdot |E'| \cdot |T|)$. The space complexity remains the same as before.

4.4 The Valid-reachability Algorithm: An Edge-optimal Algorithm

As shown in Figure 5, the full-reachability algorithm is not edge-optimal. Consider the path

$$p = [U, \varepsilon, A, t, A, f, V]$$

in graph G, which is in $I(G, G')$ and has coverage $E_p = \{(U, \varepsilon), (A, t), (A, f)\}$. Every path in G that covers exactly these transitions is in $I(G, G')$. Nonetheless, the projection of E_p onto G'' yields a graph in which *reject* is reachable from (U, U) via the path

$$[U, \varepsilon, A, f, reject]$$

However, notice that for any path in graph G that includes both the transitions (A, t) and (A, f), the first occurrence of the transition (A, t) in the path must occur before the first occurrence of (A, f). Therefore, all paths in P_p must have this property, since by definition they cover (A, t), (A, f), and (U, ε). While the set of transitions in the path by which *reject* is reachable in G''_p includes (U, ε) and (A, f), it does not include (A, t) before (A, f). So, this path cannot be in P_p and should be ignored.

The problem then is that the full-reachability algorithm considers paths that are not in P_p but reach *reject* in G''_p. By refining the notion of reachability, we arrive at an edge-optimal algorithm. We define a partial order on the edges of graph G as follows:

> $e \sqsubseteq f$ iff for all complete paths p containing both edges e and f, the first instance of e in p precedes the first instance of f in p.

We leave it to the reader to prove that $\sqsubseteq$ is indeed a partial order (it is anti-symmetric, transitive, and reflexive). An equivalent but constructive definition of $\sqsubseteq$ follows:

> $e \sqsubseteq f$ iff e dominates f[8] or (f is reachable from e, and e is not reachable from f).

The $\sqsubseteq$ relation for graph G in Figure 5 is $(U, e) \sqsubseteq (A, t) \sqsubseteq (A, f)$.

The valid-reachability algorithm is based on the following observation: If a path q contains a transition $f \in E_p$ but does not contain a transition $e \in E_p$ such that $e \sqsubseteq f$ in G, then any path with q as a prefix cannot be a member of P_p. We say that such a path does not respect $\sqsubseteq$.

The valid-reachability algorithm first checks if *reject* is reachable from (s, s') in G''_p. If not, then p is in $I(G, G')$, as before. If (s, s') is reachable, the algorithm computes R'', the set of transitions in G''_p that are reachable from (s, s') and from which *reject* is reachable. It also computes the projection R of these transitions onto G. That is,

$$R = \{(v, l) \mid ((v, v'), l) \in R''\}$$

R is a subset of E_p. If E_p contains edges e and f such that $e \notin R$, $f \in R$ and $e \sqsubseteq f$, then the algorithm outputs that p is in $I(G, G')$. Otherwise, the algorithm conservatively assumes that p is in $D(G, G')$.

It is straightforward to show that the valid-reachability algorithm is safe. The following theorem shows that it is also edge-optimal:

Theorem 3 *Given graphs G and G' and their intersection graph G''. If $P_p \subseteq I(G, G')$ for any complete path p in G, then either*

- *reject is not reachable from (s, s') in G''_p, or*
- $\exists \{e, f\} \subseteq E_p$ *s.t.* $e \sqsubseteq f, e \notin R, f \in R$

Proof: If *reject* is not reachable in G''_p then we are done. Instead, suppose that *reject* is reachable from (s, s') in G''_p. Furthermore, assume that for all $f \in R$ and $e \in E_p$, if $e \sqsubseteq f$ in G then $e \in R$. Given these assumptions, we will show that there is a complete path q in $D(G, G')$ such that $E_q = E_p$, contradicting our initial assumption that all paths with edge coverage equal to E_p are in $I(G, G')$.

There are two parts to the proof: 1. show that there is a path q_1 in G from *entry* to v that covers only transitions from R, respects $\sqsubseteq$ and

[8] An edge e dominates edge f in graph G if every path from s to f in G contains e.

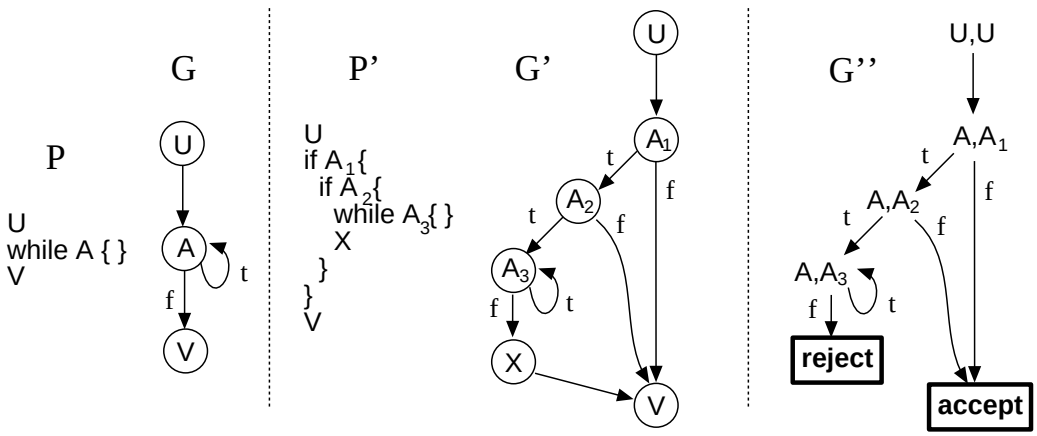

Figure 6: An example for which any CRTS algorithm based on edge coverage cannot distinguish a path in $I(G,G')$ from a path in $D(G,G')$.

induces a path in G'' from (s,s') to *reject*; 2. show that there is a path q_2 from v to x in G that covers the transitions in $E_p - E_{q_1}$ and does not cover a transition outside E_p. The concatenation of paths q_1 and q_2 yields a path q in $D(G,G')$ such that $E_q = E_p$.

The existence of path q_1 follows from the closure property of R with respect to $\sqsubseteq$ (if $f \in R$, $e \in E_p$ and $e \sqsubseteq f$ then $e \in R$), and the fact that R is the projection of R'', the transitions by which *reject* is reachable from (s,s') in G''_p.

We now show the existence of path q_2. Let e be the last edge in path q_1. Since E_p is the edge coverage of a complete path p, it follows that for all edges e and f in E_p, either f is reachable from e in G via transitions in E_p or e is reachable from f via transitions in E_p. Since the path q_1 respects $\sqsubseteq$, it also follows that for all edges f in $E_p - E_{q_1}$, either $e \sqsubseteq f$ or e and f cannot be related by $\sqsubseteq$. In the former case, f is reachable from e via transitions in E_p. In the latter case, edges e and f are not related by $\sqsubseteq$, so it follows that e and f must both be reachable from the other via transitions in E_p, completing our proof.

The time complexity of the valid-reachability algorithm is $O(|E| \cdot |E'| \cdot |T|)$. The algorithm requires, for each test in T, the construction of G''_p and the set R, which takes time $O(|E| \cdot |E'|)$, dominating all other steps in the algorithm. Using an extended version of the Lengauer/Tarjan immediate dominator algorithm [8], the immediate $\sqsubseteq$ relation for G can be computed in near-linear time and space in the size of G. To determine whether or not the set of edges R is closed with respect to E_p and $\sqsubseteq$ requires the following steps: 1. projecting E_p onto $\sqsubseteq$ to create $\sqsubseteq_p$, an $O(E)$ operation; 2. visiting each immediate relation $e \sqsubseteq_p f$ to check if $e \notin R$ and $f \in R$. As two constant-time set membership operations are performed for each immediate $\sqsubseteq_p$ relation, of which there are $O(E)$, this step takes $O(E)$ time. The space complexity of the valid-reachability algorithm remains at $O(|V| \cdot |V'|)$.

5 CRTS Using Path Coverage

Figure 6 shows that any CRTS algorithm based on edge coverage can be forced to make an incorrect (but safe) decision. It presents two programs, their graphs G and G', and their intersection graph G''. The path

$$p = [U, \varepsilon, A, t, A, f, V]$$

is in $I(G,G')$. The path p has $E_p = \{(U,\varepsilon),(A,t),(A,f)\}$. This is exactly the same set of edges covered by any path in $D(G,G')$, such as:

$$q = [U, \varepsilon, A, t, A, t, A, f, V]$$

Thus, it is impossible to determine whether a path in $I(G,G')$ or in $D(G,G')$ produced the edge set E_p.

We consider how the path profiling technique of Ammons/Ball/Larus (ABL) [2] applied to the graphs in Figure 6 can separate the paths p and q. The ABL algorithm decomposes a control flow graph into acyclic paths based on the backedges identified by a depth-first search from s. Suppose that $v \to w$ is a backedge. The ABL decomposition yields four classes of paths:

(1) A path from s to x.
(2) A path from s to v, ending with backedge $v \to w$.
(3) A path from w to v (after execution of backedge $v \to w$) ending with execution of backedge $v \to w$.
(4) After execution of backedge $v \to w$, a path from w to x.

Graph G has backedge $A \to_t A$. Applying the ABL decomposition to graph G in Figure 6 yields a total of four paths (corresponding to the four types listed above):

$$\begin{array}{lclcl} p_1 & = & [U,\varepsilon,A,f,V] & \quad p_3 & = & [A,t,A] \\ p_2 & = & [U,\varepsilon,A,t,A] & \quad p_4 & = & [A,f,V] \end{array}$$

The ABL algorithm inserts instrumentation into program P to track whether or not each of these four paths is covered in an execution. Recall the paths p and q that got edge-based CRTS into trouble. Path p is composed of the paths p_2 followed by p_4, so ABL will record that only these two paths are covered when p executes. On the other hand, the path q is composed of p_2, followed by p_3, followed by p_4. Thus, for this example where edge coverage could not distinguish the two paths, the ABL path coverage does.

As mentioned in the introduction, an adversary can create a graph G' such that any control-flow-based RTS algorithm that records less than the complete path executed through G will be unable to distinguish a path in $I(G,G')$ from a path in $D(G,G')$. This is only true if G contains cycles, as it does in our example

In the example from Figure 6, we can defeat the ABL path coverage by adding another **if-then** conditional (with basic block A) around the outermost conditional in program P'. Now, the path

$$q = [U, \varepsilon, A, t, A, t, A, f, V]$$

is in $I(G,G')$ and the path in which (A,t) occurs one more time,

$$r = [U, \varepsilon, A, t, A, t, A, t, A, f, V]$$

is in $D(G,G')$. However, both these paths cover exactly the set of ABL paths $\{p_2, p_3, p_4\}$, so they will not be distinguished unless longer paths are tracked. For any cutoff chosen, we can add another level of nesting and achieve the same effect.

6 Related Work

Rothermel and Harrold define a framework for comparing different regression test selection methods [11], based on four characteristics:

- *Inclusiveness*, the ability to choose modification revealing tests (paths in $D(G,G')$);
- *Precision*, the ability to eliminate or exclude tests that will not reveal behavioral differences (paths in $I(G,G')$);
- *Efficiency*, the space and time requirements of the method, and
- *Generality*, the applicability of the method to different classes of languages, modifications, etc.

Our approach shares many similarities with the RH algorithm. The three reachability algorithms are based on control flow analysis and edge coverage. The partial-reachability algorithm is just as inclusive as the RH algorithm but is more precise with equivalent efficiency. The full-reachability and valid-reachability algorithms are even more precise, but at a greater cost. We have not yet considered how to generalize our algorithms to handle interprocedural control flow, as they have done.

Rothermel shows that the problem of determining whether or not a new program is "modification-traversing" with respect to an old program and a test t is PSPACE-hard [10]. Intuitively, this is because the problem involves tracing the paths that the programs execute and the paths can have size exponential in the input program size (or worse). Of course, given a complete path through an old program and a new program, it is a linear-time decision procedure to determine if the new program contains the path. However, this defines away the real problem: that the size of the path can be unbounded. We have considered the best a CRTS algorithm can do when the amount of information recorded about a program's execution is $O(E)$ (edge coverage) or exponential in the number of edges (ABL path coverage).

7 Summary

We have formalized control-flow-based regression test selection using finite automata theory and the intersection graph. The partial-reachability algorithm has time and space complexity equivalent to the best previously known algorithm, but is more precise. In addition, we defined a strong optimality condition for edge-based regression test selection algorithms and demonstrated an algorithm (valid-reachability) that is edge-optimal. Finally, we considered how path coverage can be used to further improve regression test selection.

A crucial question on which the practical relevance of our work hinges is whether or not the "multiply-visited" vertex condition defined by Rothermel and Harrold occurs in practice. For versions of programs that do not have this condition, the RH algorithm is optimal. When this condition does occur, as we have shown, the RH algorithm is not even edge-optimal. We plan to analyze the extensive version control repositories of systems in Lucent [3] to address this question.

Acknowledgements

Thanks to Mooly Sagiv and Patrice Godefroid for their suggestions pertaining to finite state theory. Thanks also to Glenn Bruns, Mary Jean Harrold, Gregg Rothermel, Mike Siff, Mark Staskauskas and Peter Mataga for their comments.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, June 1997. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation.

[3] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk... In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[6] Y.-F. Chen, D. Rosenblum, and K. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, 1994.

[7] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[8] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[9] T. Ostrand and E. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the 6th Annual Pacific Northwest Software Quality Conference*, pages 233–247, September 1988.

[10] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. Ph.D. thesis, Clemson University, December 1995.

[11] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[12] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.