Structural Testing Based on Minimum Kernels

Elena Dubrova
Royal Institute of Technology, IMIT/KTH, 164 46 Kista, Sweden

Abstract

Structural testing techniques, such as statement and branch coverage, play an important role in improving dependability of software systems. However, finding a set of tests which guarantees high coverage is a time-consuming task. In this paper we present a technique for structural testing based on kernel computation. A kernel satisfies the property that any set of tests which executes all vertices (edges) of the kernel executes all vertices (edges) of the program’s flowgraph. We present a linear-time algorithm for computing minimum kernels based on pre- and post-dominator relations of a flowgraph.

1 Introduction

Software testing is the process of executing a program with the intent of finding errors [1]. Testing is a major consideration in software development. In many organizations, more time is devoted to testing than to any other phase of software development. On complex projects, test developers might be twice or three times as many as code developers on a project team.

Thorough software testing is required for improving dependability of real-time computing applications using software-embedded systems. A number of serious accidents in the past has been caused by incomplete software testing, including the Therac-25 radiation overdoses [2], the crash of the British destroyer Sheffield [3], the accidents of Ariane 5 [4] and Airbus A320-211 [5].

There are two types of software testing: functional and structural. Functional testing (also called behavioral testing, black-box testing, closed-box testing), compares test program behavior against its specification. Structural testing (also called white-box testing, glass-box testing) checks the internal structure of a program for errors. For example, suppose we test a program which adds two integers. The goal of functional testing is to verify whether the implemented operation is indeed addition instead of e.g. multiplication. Structural testing does not question the functionally of the program, but checks whether the internal structure is consistent. A strength of the structural approach is that the entire software implementation is taken into account during testing, which facilitates error detection even when the software specification is vague or incomplete.

The effectiveness of structural testing is normally expressed in terms of test coverage metrics, which measure the fraction of code exercised by test cases. Common test coverage metrics are statement, branch, and path coverage [1]. Statement coverage requires that the program under test is run with enough test cases, so that all its statements are executed at least once. Decision coverage requires that all branches of the program are executed at least once. Path coverage requires that each of the possible paths through the program is followed. Path coverage is the most reliable metric, however, it is not applicable to large systems, since the number of paths is exponential to the number of branches.

In this paper we present a technique for structural testing which finds a part of program’s flowgraph, called kernel, with the property that any set of tests which executes all vertices (edges) of the kernel executes all vertices (edges) of the flowgraph. We present an linear-time algorithm for computing kernels of a minimum size based on pre- and post-dominator relations of a flowgraph.

A related previous work is an algorithm presented in [6]. First, it constructs pre- and post-dominator trees of the program’s flowgraph. Then, both trees are combined and the resulting graph is reduced by merging its strongly connected components and eliminating composite edges implied by dominator transitivity. The obtained super block dominator graph represents kernels of the flowgraph.

The main difference between the presented algorithm and the algorithm [6] is that we compute only one minimum kernel for a given flowgraph instead of all possible ones, which clearly can be done with less computational effort. The presented $O(|V| + |E|)$ algorithm computes minimum kernels directly from pre- and post-dominator trees of a flowgraph with $|V|$ vertices and $|E|$ edges, without constructing the super block dominator graph. We prove that the resulting kernel is minimum. No proof of minimality is given in [6].

Other related works include: Bertolino and Marre’s algorithm [7] for finding path covers in a flowgraph, in
which unconstrained arcs are analogous to the leaves of the
dominator tree; Ball’s [8] and Podgurski’s [9] techniques
for computing control dependence regions in a flowgraph,
which are similar to the super blocks of [6]; Agarwal’s algo-
rithm [10], which addresses the coverage problem at an inter-
procedural level.

The paper is organized as follows. Section 2 gives an
overview of coverage techniques. Section 3 states the pre-
liminaries. In Section 4, kernels are introduced. Section
5 presents an algorithm for computing minimum ker-
nels. Section 6 shows use of kernels in branch coverage.
Section 7 concludes the paper.

2 Statement and Branch Coverage

In this section, we give a brief overview of statement and
branch coverage techniques.

2.1 Statement Coverage

Statement coverage (also called line coverage, segment
coverage [11], C1 [1]) examines whether each executable
statement of a program is followed during a test. An exten-
sion of statement coverage is basic block coverage, in which
each sequence of non-branching statements is treated as one
statement unit.

The main advantage of statement coverage is that it can
be applied directly to object code and does not require pro-
cessing source code. The disadvantages are:

• Statement coverage is insensitive to some control
structures, logical AND and OR operators, and switch
labels.

• Statement coverage only checks whether the loop body
was executed or not. It does not report whether loops
reach their termination condition. In C, C++, and
Java programs, this limitation affects loops that con-
tain break statements.

As an example of the insensitivity of statement coverage
to some control structures, consider the following code:

\[
\begin{align*}
x &= 0; \\
\text{if (condition)} & \quad x = x + 1; \\
y &= 10/x;
\end{align*}
\]

If there is no test case which causes condition to
evaluate false, the error in this code will not be detected
in spite of 100% statement coverage. The error will ap-
pear only if condition evaluates false for some test case.
Since if-statements are common in programs, this problem
is a serious drawback of statement coverage.

2.2 Branch Coverage

Branch coverage (also referred to as decision coverage,
all-edges coverage [12], C2 [1]) requires that each branch
of a program is executed at least once during a test. Boolean
expressions of if- or while-statements are checked to be
evaluated to both true and false. The entire Boolean expres-
sion is treated as one predicate regardless of whether it con-
tains logical AND and OR operators. Switch statements,
exception handlers, and interrupt handlers are treated simi-
larly. Decision coverage includes statement coverage since
executing every branch leads to executing every statement.

An advantage of branch coverage is its relative sim-
plectic. It allows overcoming many problems of statement
coverage. However, it might miss some errors as demon-
strated by the following example:

\[
\begin{align*}
\text{if (condition1)} & \quad x = 0; \\
\text{else} & \quad x = 2; \\
\text{if (condition2)} & \quad y = 10* x; \\
\text{else} & \quad y = 10/x;
\end{align*}
\]

The 100% branch coverage can be achieved by
two test cases which cause both condition1 and
condition2 to evaluate true, and both condition1
and condition2 to evaluate false. However, the er-
ror which occurs when condition1 evaluates true and
condition2 evaluates false will not be detected by these
two tests.

The error in the example above can be detected by exer-
cising every path through the program. However, since the
number of paths is exponential to the number of branches,
testing every path is not possible for large systems. For
example, if one test case takes \(0.1 \times 10^{-5}\) seconds to ex-
cute, then testing all paths of a program containing 30 if-
statements will take 18 minutes and testing all paths of a
program with 60 if-statements will take 366 centuries.

Branch coverage differs from basic path coverage,
which requires each basis path in the program flowgraph
to be executed during a test [13]. Basis paths are a mini-
mal subset of paths that can generate all possible paths by
linear combination. The number of basic paths is called the
cyclomatic number of the flowgraph.

3 Preliminaries

A flowgraph is a directed graph \(G = (V, E, entry, exit)\),
where \(V\) is the set of vertices representing basic blocks of
the program, \(E \subseteq V \times V\) is the set of edges connecting the
Algorithm Example

```plaintext
algorithm Example
b1;
while(b2) {
  for(b3) {
    b4;
    for(b5) {
      if(b6) b7;
      else b8;
    }
    if(b9) break;
  }
  switch(b10) {
    case 1: while(b11) b12;
    case 2: if(b13) b14;
    else continue;
    default: b15;
    break;
  }
  b16;
} b17;
end
```

Figure 1: Example C program.

Vertices, and entry and exit are two distinguished vertices of V. Every vertex in V is reachable from entry vertex, and exit is reachable from every vertex in V.

Figure 2 shows the flowgraph of the C program in Figure 1, where b1, b2, ..., b17 are blocks whose contents are not relevant for our purposes.

A vertex v pre-dominates another vertex u, if every path from entry to u contains v. A vertex v post-dominates another vertex u, if every path from u to exit contains v.

By Pre(v) and Post(v) we denote sets of all nodes which pre-dominate and post-dominate v, respectively. E.g. in Figure 2, Pre(5) = {1, 2, 3, 4} and Post(5) = {9, 10, 17}.

Many properties are common for pre- and post-dominators. Further in the paper, we use the word dominator to refer to cases which apply to both relationships.

Vertex v is the immediate dominator of u, if v dominates u and every other dominator of u dominates v. Every vertex v ∈ V except entry (exit) has a unique immediate pre-dominator (post-dominator), idom(v) [14]. For example, in Figure 2, vertex 4 is the immediate pre-dominator of 5, and vertex 9 is the immediate post-dominator of 5. The edges (idom(v), v) form a directed tree rooted at entry for pre-dominators and at exit for post-dominators. Figures 3 and 4 show the pre- and post-dominator trees of the flowgraph in Figure 2.

The problem of finding dominators was first considered in late 60’s by Lorry and Medlock [14]. They presented an O(|V|^4) algorithm for finding all immediate dominators in a flowgraph. Successive improvements of this algorithm were done by Aho and Ullman [15], Purdom and Moore [16], Tarjan [17], and Lengauer and Tarjan’s [18]. Lengauer and Tarjan’s algorithm [18] is a nearly-linear algorithm with the complexity O(|E|α(|E|, |V|)), where α is the standard functional inverse of the Ackermann function. Linear algorithms for finding dominators were presented by Harel [19], Alstrup et al. [20], and Buchsbaum et al. [21].

4 Statement Coverage Using Kernels

In this section we present a technique for finding a subset of the program’s flowgraph vertices, called kernel, with the property that any set of tests which executes all vertices of the kernel executes all vertices of the flowgraph. A 100% statement coverage can be achieved by constructing a set of tests for the kernel.

Definition 1 A vertex v ∈ V of the flowgraph is covered by a test case t if the basic block of the program representing v is reached at least once during the execution of t.

The following Lemma is the basic property of our technique, as well as the technique presented in [6].

Lemma 1 If a test case t covers u ∈ V, then it covers any post-dominator of u as well:

(t covers u) ∧ (v ∈ Post(u)) ⇒ (t covers v).

Proof: If v post-dominates u, then every path from u to exit contains v. Therefore, if u is reached at least once during the execution of t, then v is reached, too.

□
Theorem 1 The set \( L_{post} - L^D_{post} \) is a minimum kernel.

Proof: Lemma 1 shows that, if a vertex of a flowgraph is covered by a test case \( t \), then all its post-dominators are also covered by \( t \). Therefore, in order to cover all vertices of a flowgraph, it is sufficient to cover all leaves \( L_{post} \) in its post-dominator tree, i.e. \( L_{post} \) is a kernel.

\( L^D_{post} \) contains all vertices of \( L_{post} \) which pre-dominate some vertex of \( L_{post} \). If \( v \) is a pre-dominator of \( u \), and \( u \) is covered by \( t \), then \( v \) is also covered by \( t \), since every path from \( entry \) to \( u \) contains \( v \) as well. Thus, any set of tests which covers \( L_{post} - L^D_{post} \), covers \( L_{post} \) as well. Since \( L_{post} \) is a kernel, \( L_{post} - L^D_{post} \) is a kernel, too.

Next, we prove that the set \( L_{post} - L^D_{post} \) is a minimum kernel. Suppose that there exists another kernel, \( K' \), such that \( |K'| < |L_{post} - L^D_{post}| \). If \( v \in K' \) and \( v \notin L_{post} \), then \( v \in Post(u) \) for some \( u \in L_{post} \). Since every path from \( u \) to \( exit \) contains \( v \), if \( u \) is reached at least once during the execution of some test case, then \( v \) is reached, too. Therefore, \( K' \) remains a kernel if we replace \( v \) by \( u \).

Suppose we replaced all \( v \in K' \) such that \( v \notin L_{post} \) by \( u \in L_{post} \) such that \( v \in Post(u) \). Now, \( K' \subseteq L_{post} \). If there exists \( w \in L_{post} - K' \) such that, for all \( u \in K' \), \( w \notin Pre(u) \) then there exists at least one path from \( entry \) to each \( u \) which does not contain \( w \). This means that there exists a test set, formed by the set of paths \( path(u) \) where \( path(u) \) is the path to \( u \) which does not contain \( w \), that covers \( K' \) but not \( w \). According to Definition 2, this implies that \( K' \) is a kernel. Therefore, to guarantee that \( K' \) is a kernel, \( w \) must be a pre-dominator of some \( u \in K' \), for all \( w \in L_{post} - K' \). This implies that \( |K'| = |L_{post} - L^D_{post}| \).

\[ |L_{post} - L^D_{post}| = |L_{pre} - L^D_{pre}| \]

The next theorem shows that the set \( L_{pre} - L^D_{pre} \) is also a minimum kernel.

Theorem 2

In this section, we present an linear-time algorithm for computing minimum kernels. The pseudo-code is shown in Figure 5.

First, pre- and post-dominator trees of the flowgraph \( G = (V,E,entry,exit) \), denoted by \( T_{pre} \) and \( T_{post} \), are computed. Then, the numbers of leaves of the trees, \( L_{pre} \) and \( L_{post} \), are compared. According to Theorems 1 and 2, both, \( L_{post} - L^D_{post} \) and \( L_{pre} - L^D_{pre} \), represent minimum kernels.
The procedure FINDLD is applied to the smaller of the sets $L_{pre}$ and $L_{post}$.

FINDLD checks whether the leaves $L_{pre}$ of the tree $T_{pre}$ are dominated by some vertex of $L_{pre}$ in another tree, $T_{post}$, or vice versa. In other words, FINDLD computes the set $L_{pre}^D (L_{post}^D)$.

**Theorem 3** The algorithm KERNEL computes a minimum kernel of a flowgraph $G = (V, E, entry, exit)$ in $O(|V| + |E|)$ time.

**Proof:** The correctness of the algorithm follows directly from Theorems 1 and 2. The complexity of the KERNEL is determined by the complexity of computing the $T_{pre}$ and $T_{post}$ trees. A dominator tree can be computed in $O(|V| + |E|)$ time [20]. Thus, the overall complexity is $O(|V| + |E|)$.

Figure 5: Pseudo-code of the algorithm for computing minimum kernels.

As an example, let us compute a minimum kernel for the flowgraph in Figure 2. Its pre- and post-dominator trees are shown in Figures 3 and 4. $T_{pre}$ has 7 leaves, $L_{pre} = \{7, 8, 12, 14, 15, 16, 17\}$, and $T_{post}$ has 9 leaves, $L_{post} = \{1, 3, 6, 7, 8, 11, 13, 14, 15\}$. So, we check which of the leaves of $T_{pre}$ dominates at least one other leaf of $T_{pre}$ in $T_{post}$. Leaves $L_{pre}$ are marked as shaded circles in $T_{post}$ in Figure 4. We can see that, in $T_{post}$, vertex 16 dominates 12 and 14, and vertex 17 dominates all other leaves of $L_{pre}$. Thus, $L_{pre}^D = \{16, 17\}$. The minimum kernel $L_{pre} - L_{pre}^D$ consist of five vertices: 7, 8, 12, 14 and 15.

For a comparison, let us compute the minimum kernel given by $L_{post} - L_{post}^D$. The $L_{post}$ leaves are marked as shaded circles in $T_{pre}$ in Figure 3. We can see that, in $T_{pre}$, vertex 6 dominates 7 and 8, vertex 3 dominates 6, 7 and 8, vertex 13 dominates 14, and vertex 1 dominates all other leaves of $L_{post}$. Thus, $L_{post}^D = \{1, 3, 6, 13\}$. The minimum kernel $L_{post} - L_{post}^D$ consist of five vertices: 7, 8, 11, 14 and 15. It is easy to see from the flowgraph why vertices 11 and 12 are interchangeable in two minimum kernels.

Figure 6: Edge pre-dominator tree of the flowgraph in Figure 2; shaded vertices are leaves of the tree in Figure 7.

Figure 7: Edge post-dominator tree of the flowgraph in Figure 2; shaded vertices are leaves of the tree in Figure 6.

### 6 Decision Coverage Using Kernels

The kernel-based technique described above can be similarly applied to branch coverage by constructing pre- and post-dominator trees for the edges of the flowgraph instead of for its vertices. Figures 6 and 7 show edge pre- and post-dominator tree of the flowgraph in Figure 2.

Similarly to Definition 2, a kernel set for edges is defined as a subset of edges of the flowgraph which satisfies the property that any set of tests which executes all edges of the kernel executes all edges of the flowgraph. A 100% branch coverage can be achieved by constructing a set of tests for the kernel. Minimum kernels for Figures 6 and 7 are: $L_{pre} - L_{pre}^D = \{i, h, k, p, q, t, v, w, x, y\}$ and $L_{post} - L_{post}^D = \{f, g, k, n, p, q, t, w, x, y\}$.
7 Conclusion

In this paper we present a technique for structural testing based on kernel computation. A kernel has the property that any set of tests which executes all vertices (edges) of the kernel executes all vertices (edges) of the program’s flowgraph. For large programs, the number of statements and branches to be covered by tests might be prohibitive. Kernels allow us to reduce the amount of test cases to be constructed while preserving a 100% coverage. We present a linear-time algorithm for computing kernels of a minimum size based on pre- and post-dominator relations of a flowgraph.

Future work includes investigating how properties of dominators can be further exploited. One possibility is to apply the presented algorithm to testing of finite state machines (FSMs). Similarly to program coverage, a state or a transition of the FSM is considered covered if it is visited during the execution of the input test sequence [22]. Dominator relations can be used to identify kernels for FSM’s states/transitions.

Acknowledgments

I am indebted to the anonymous reviewer who gave many valuable comments over the manuscript and suggested improvements in the proof of the Theorem 1.

This work was supported in part by the Research Grant 2002-4300 from the Swedish Research Council Vetenskapsrådet.

References