Advance Access published on May 3, 2005
doi:10.1093/comjnl/bxh093

Branch-Coverage Testability Transformation for Unstructured Programs

R. M. HIERONS[1], M. HARMAN[2] AND C. J. FOX[3]

[1]*Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex UB8 3PH, UK*
[2]*Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK*
[3]*Department of Computer Science, University of Essex, Colchester CO4 3SQ, UK*
Email: rob.hierons@brunel.ac.uk

Test data generation by hand is a tedious, expensive and error-prone activity, yet testing is a vital part of the development process. Several techniques have been proposed to automate the generation of test data, but all of these are hindered by the presence of unstructured control flow. This paper addresses the problem using testability transformation. Testability transformation does not preserve the traditional meaning of the program, rather it deals with preserving test-adequate sets of input data. This requires new equivalence relations which, in turn, entail novel proof obligations. The paper illustrates this using the branch coverage adequacy criterion and develops a branch adequacy equivalence relation and a testability transformation for restructuring. It then presents a proof that the transformation preserves branch adequacy.

Received 29 April 2004; revised 24 January 2005

1. INTRODUCTION

Testing is an important part of the software verification process. However, it is also an expensive process, often taking up ∼50% of the cost of development [1].

When generating tests, it is normal to use a test criterion that states what it means for a set of test inputs to be sufficient (or 'adequate' in the nomenclature of software testing). There are two main classes of test criteria: white-box criteria, which are based on the structure of the code, and black-box criteria, which are based on the specification. Given a test criterion, the problem is to find test data that satisfy this criterion.

Producing test data by hand is tedious, expensive and error-prone. For these reasons, automated test data generation has remained a topic of interest for the past three decades. Several techniques for automated test data generation have been proposed, including symbolic execution [2, 3, 4], constraint solving [5, 6] and search-based testing [7, 8, 9, 10, 11, 12, 13, 14].

Unstructured programs present a problem for all of these automated test data generation techniques. In this paper, an unstructured program is defined as one which contains any form of explicit jump statement, typically represented by `goto`, `exit`, `break` or `continue` statements. This is contrasted with a structured program in which the only control flow that is permitted is that which is expressed by `if-then-else-fi` conditionals, single-exit `while-do-od loops` and statement sequencing.

Early approaches [2, 4] to test data generation used symbolic execution. The symbolic execution technique seeks to symbolically execute the program to allow the test data generator to express constraints on predicates to be executed in terms of initial state variables.

In the 1990s constraint solving techniques were developed [5, 15]. The constraint solving technique is, in essence, a mirror image of the symbolic execution approach; it seeks to push the constraints on predicates back to the initial statement of the program, at which point they are transformed into constraints upon the input of the program.

Both symbolic execution and constraint solving face problems in the presence of loops, because it is difficult to determine a precise transformation to apply to a set of constraints in order to move it through a loop boundary. These problems with loops are further compounded if the program to be tested is unstructured because, in unstructured programs, there is the additional problem of determining where the loop boundaries are, although it is possible to determine the overall control flow structure [16].

To overcome difficulties caused by loops for constraint-based and symbolic-execution-based test data generation, more recent work has focused on the use of search-based techniques such as hill climbing [7], simulated annealing [14] and genetic algorithms [8, 9, 10, 11, 12, 13]. All these search-based techniques view the space of inputs to the program under test as a search space. The criterion which expresses adequacy of sets of test inputs guides the automated search for such sets.

In the chaining method [7], data flow information is used to locate previous definitions of variables mentioned in a ‘problem predicate’. A problem predicate is one for which test data that forces the predicate to select a particular branch is yet to be found. Hill climbing is then applied to attempt to find input values that force the problem predicate to achieve the required value through a given set of definitions of variables mentioned in this predicate. If this fails, a different set of definitions is chosen or the attempt is abandoned. Work on the chaining method has hitherto focused on achieving full statement coverage but it has been argued that this approach could be extended to other test criteria [7].

In the simulated annealing and the genetic algorithms methods, the test criterion is transformed to a fitness function which guides the search for test data. In the case of branch coverage (and related structural criteria), the fitness function must capture how close a candidate input comes to executing the desired branch. The success of these approaches in finding test data for real systems has led to them receiving much attention. However, the definition of the fitness function requires control dependence analysis, which is notoriously problematic in the presence of poor structure [17, 18, 19, 20]. The presence of many `goto`s makes it difficult to determine how close a test input comes to hitting a desired branch, because the flow of control is not evident. Consider the program below.

```
while p do
  ...
  if q then exit 1 fi;
  ...
od;
if p then S1 else S2 fi
```

In this program, one target node is the statement `S2` which occurs after the end of the `while` loop. Notice that there are (apparently) two ways execution can reach `S2`. The loop may terminate ‘normally’, because the predicate `p` becomes `false`, or it may terminate ‘abnormally’ through the `exit` statement in the body of the loop. Now, in the case of this program, should the loop terminate normally, it will be impossible for `S2` to be executed, since all paths which involve normal termination and execution of `S2` are infeasible. Therefore, the only way in which `S2` could possibly be executed would be for the loop to terminate through the `exit` statement.

Unfortunately, it is difficult to determine which paths are feasible. In a test data generation system, the program will be instrumented to compute values which track the execution of the program. The test data generator will attempt to force the loop to terminate, either through the `exit` statement or through ‘normal’ termination. However, the test data generator will not (and cannot) know which of the two ways of terminating is more likely to lead to the execution of the desired target node, `S2`. In extreme cases, such as this one, where one termination route is infeasible, the test data generator may spend all its efforts attempting to generate a test input which drives the computation down an infeasible path.

Even in less extreme cases, it is possible for the test data generator to waste effort because two distinct paths must be optimized, rather than one. If the program is restructured, so that there is only one way of terminating the loop, then this possibility is eliminated.

Current techniques and tools for automatically generating test data, to satisfy white-box test criteria, are only applicable to structured code. Thus, the applicability of all such tools and techniques will be extended by the development of transformations that take unstructured codes and return structured codes from which we can generate test data.

Traditional work on transformation of unstructured programs is concerned largely with making the program easier to understand [21], or with compiler optimizations and parallelizations [22]. Of course, the resultant structured programs have other advantages, such as lending themselves to compositional analysis and abstract interpretation, without recourse to continuation-based semantics. They also support efficient program-dependent graph generation.

In general, the goal of such transformations is to remove `goto` statements and create, where possible, a single-entry, single-exit control flow structure. In order to transform unstructured programs to improve test data generation, it is also important to produce single-entry, single-exit control flow. However, as will be seen, the goal of improving testability, rather than comprehension or compiler optimizations, leads to very different transformation algorithms, equivalence relations and (consequently) proof obligations.

Although there are several kinds of statements that give rise to unstructured programs, this paper will focus on the `exit` (or `break`) statements, which are a common source of unstructuredness. The paper examines the problem of transforming a program p to form a structured program p' such that any set of test inputs that provides 100% branch coverage for p' also provides 100% branch coverage for p.

In Section 4, we show that there are programs with `exit` statements that are not path equivalent to any structured program. Thus, we cannot expect any algorithm that removes `exit` statements to preserve path equivalence. Similarly, it is clear that we need something other than functional equivalence: we need to preserve elements of the structure to ensure that the meaning of the adequacy criterion remains unchanged after transformation.

Our approach uses testability transformation [23]. A testability transformation does not need to preserve the traditional meaning of the program to be transformed, rather it preserves the sets of test inputs which are adequate according to the test criterion. In our case, we are dealing with branch-coverage preserving transformations. Motivated by this, we define a new form of equivalence called branch-coverage equivalence (defined in Section 5). This new form of equivalence gives rise to new transformations and thus fresh ‘proof of correctness’ obligations.

We then give a transformation algorithm that takes a program p with multiple-level `exit` statements[1] and returns a branch-coverage equivalent program p' such that any set T of test inputs that provides 100% branch coverage for p' is guaranteed to give 100% branch coverage for p. A proof of correctness is given in terms of preservation of branch-coverage adequate test data.

The rest of this paper is structured as follows. Section 2 describes control flow graphs (CFGs) and two notions of program equivalence, and restates the notion of a structured program. Section 3 describes `exit` statements and branch coverage. Section 4 describes related work. Section 5 defines what it means for two programs to be branch-coverage equivalent. Section 6 introduces a transformation algorithm and proves that it is correct. Finally, Section 7 draws conclusions and discusses future work.

2. PRELIMINARY DEFINITIONS

2.1. The syntax

This section will briefly outline the syntax discussed in this paper. We will assume that the problem is to take a single procedure or function that contains one or more `exit` statements and transform it into a code that contains no `exit` statements.

This paper will focus on one common construct that leads to unstructured programs: the use of `exit` statements in loops. An `exit` statement leads to the flow of control leaving a loop. The presence of an `exit` thus provides different ways by which the flow of control may leave a loop. Some programming languages allow multiple-level `exit` statements.

The only control structures considered in this paper are `while-do-od` loops and `if-then-else-fi` statements. This simplifies the exposition. However, it will become clear that the approach may be extended to other control constructs.

For purposes of transformations in this paper, we 'abstract out' all information regarding assignments, input statements and output statements: these are considered to be atomic statements. Only the control structure is of relevance, as are the `exit` statements. The syntax is defined in Figure 1.

A multiple-level `exit` statement is parameterized by a non-negative integer whose value is known at compile time. The semantics are defined such that `exit` n jumps to the statement immediately after the end of the nth enclosing loop. This paper uses the Wide Spectrum Language convention [24] that when n is zero, the meaning of `exit` n is identical to `skip`; this convention reduces the number of transformation rules required to cover all cases. Such multiple-level `exit` statements are sufficiently powerful to capture the full complexity of the unstructuredness issue, since an arbitrary `goto` program can be converted into a program with multiple-level exits, without the introduction of any new variables [21].

```
⟨Program⟩      ::= ⟨Statements⟩
⟨Statements⟩   ::= ⟨Statement⟩
                 | ⟨Statement⟩;  ⟨Statements⟩
                 | skip
⟨Statement⟩    ::= ⟨Linear-block⟩
                 | if ⟨Predicate⟩
                   then ⟨Statements⟩
                   [else ⟨Statements⟩]
                   fi
                 | while ⟨Predicate⟩
                   do ⟨Statements⟩ od
⟨Linear-block⟩ ::= ⟨Non-control⟩
                 | ⟨Non-control⟩;  ⟨Linear-block⟩
⟨Non-control⟩  ::= ⟨Atomic-statement⟩
                 | exit n
⟨Predicate⟩    ::= ⟨Atomic-predicate⟩
                 | ( ⟨Predicate⟩ ∧ ⟨Predicate⟩ )
                 | ¬⟨Predicate⟩
```

FIGURE 1. Syntax of the pseudo-code. For our purposes, it is sufficient to abstract ⟨*Atomic-statement*⟩ and ⟨*Atomic-predicate*⟩ to sets of identifiers. n denotes the natural numbers. Statements of the form `exit` n must appear at an appropriate depth within nested `while` loops.

The process outlined in this paper applies to code that contains many functions and procedures. This is because it is sufficient to consider the individual functions and procedures. However, white-box test criteria, such as 100% branch coverage, are typically applied in unit testing where individual modules or classes are tested.

2.2. The control flow graph

Given a program p, the CFG for p is a directed graph G that has a set of vertices which represent the statements and predicates of p and edges between these vertices. The edges are defined by rules that aim to represent the possible flow of control. For example, in the part of the CFG that represents

```
if x>0 then
  x=x+1
else
  x=x-1
fi
```

if the predicate `x>0` is represented by vertex n_1, `x=x+1` by vertex n_2 and `x=x-1` by vertex n_3 then there is an edge from n_1 to n_2 and an edge from n_1 to n_3. In this case, there will also be exit edges from vertices n_2 and n_3 that merge, indicating a common flow of control following this statement. This is illustrated in Figure 2.

The CFG has two special vertices: the start vertex that represents the start of execution (or 'entrance' into the code block represented by the CFG) and the end vertex that represents termination (or 'exit' from the code block represented by the CFG).

[1] We assume that each loop has at most one `exit` statement. This restriction is discussed in greater detail in Section 6.

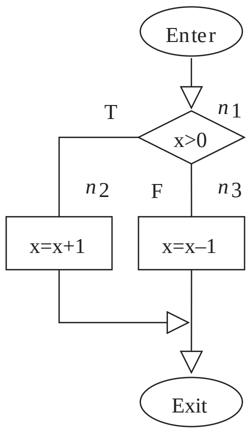


FIGURE 2. CFG for the program fragment `if x>0 then x=x+1 else x=x-1 fi`.

Note that the edges are defined using rules that abstract certain details. Thus, there may be edges in the CFG that cannot be traversed. For example, the CFG for the code

```
x=0;
if x>0 then
  x=x+1
fi
```

has an edge to the vertex representing `x=x+1` even though this statement is not reachable.

At times we have to consider paths through a program.

DEFINITION 2.1. (PATHS). *Given the CFG G for a program p, a path π is a sequence of consecutive edges of G from the start node to the end node.*

Each path π defines a corresponding path condition $c(\pi)$ of π: an input leads to the traversal of path π if and only if it satisfies $c(\pi)$. The path π is feasible if and only if there is an input that satisfies $c(\pi)$.

2.3. Forms of program equivalence

2.3.1. Functional equivalence

A program p is functionally equivalent to program p' if the external functional behaviour of the two programs is indistinguishable, so that for all inputs i, applying p to i gives the same result as applying p' to i. Functional equivalence requires identical non-termination behaviour: if p fails to terminate on input u, so does p'. It does not require equivalence of non-functional behaviour. For example, p and p' may take a different number of steps to produce the output. Crucially, functional equivalence is an extensional notion; it imposes no requirements on the internal implementation details of p and p'.

We can also define a restricted notion of functional equivalence, where the equivalence of the behaviours for a subset of the possible inputs is taken into account.

With program transformations, we may wish to preserve some aspects of the internal structure of the program. For this, we require an intensional notion of equivalence.

2.3.2. Path equivalence

One intensional aspect of a program that we may wish to preserve is the nature of the paths, or execution traces. There are various ways of defining path equivalence. We state the strictest notion, which is defined in terms of a program's CFG.

Two programs p and p' are CFG-path equivalent if for every execution trace in the programs' CFGs, the same sequence of statements is executed under the same conditions. With CFG-path equivalence there is no attempt to check which paths represent computations that can be realised: it includes paths that can never be executed because the path condition can never be met.

2.4. Structured programs

In the context of this paper, a structured program is one that does not exploit `exit` statements.

DEFINITION 2.2. (STRUCTURED PROGRAM). *A structured program is one in which the only control flow that is permitted is that which is expressed by `if-then-else-fi` conditionals, single-exit `while-do-od` loops and statement sequencing.*

All other programs are considered to be unstructured.

DEFINITION 2.3. (UNSTRUCTURED PROGRAM). *An unstructured program is one that contains one or more `exit` statements.*

Implicitly, this can be taken to include all programs containing other forms of arbitrary jump statements, such as `goto`. Such programs can be transformed into path equivalent programs containing `exit` statements without `goto` statements and additional variables [21].

Following from these definitions, a program is either structured or unstructured; for the purposes of this paper, there is no notion of one program being 'more structured' than another.

3. BRANCH COVERAGE

This paper concentrates on one commonly used notion of coverage: branch coverage. A branch is an edge in the CFG, from a node n where n has more than one edge leaving it. Branches are thus associated with constructs such as `if` statements and loops. There is an additional branch from the start node and every path passes through this branch. Branch coverage may be defined in the following manner.

DEFINITION 3.1. (BRANCH COVERAGE). *A test input t covers a branch b of program p if when p is tested with t, the flow of control passes through b. A set T of test inputs covers a branch b of program p if some $t \in T$ covers b.*

DEFINITION 3.2. (PROPORTION OF BRANCHES COVERED). *Given a set T of test inputs and program p, the branch coverage of p achieved by T, as a percentage, is:*

$$\frac{\textit{Number of branches of } p \textit{ covered by } T}{\textit{Number of feasible branches in } p} \times 100$$

There has been a significant amount of work on the problem of automatically generating a set of test inputs that provides 100% branch coverage. While this problem is generally not computable, several approaches have proved to be effective in practice [8, 9, 11, 14, 25, 26, 27].

Structural test criteria, such as branch coverage, base the requirements for testing on the structure of the code. Thus, there is a potential danger in transforming the code to make it structured: if we transform an unstructured program p to form a structured program p', a set of test inputs generated to cover the branches of p' does not need to cover the branches of p. Thus, we require a set of transformations such that:

(i) The transformations will take unstructured codes and return structured codes.
(ii) If program p is transformed into p' then any set of test inputs that gives 100% branch coverage for p' will also achieve 100% branch coverage for p.

It is requirement (ii) above which makes this problem different from the traditional problem of restructuring unstructured programs.

We require a transformation system that converts unstructured codes into structured codes but preserves something other than functional equivalence. Note that in principle it is not necessary to preserve functional equivalence. The following simple program serves as an illustration, here S_1 and S_2 contain no conditionals and thus no branches.

```
if $P$ then
  $S_1$
else
  $S_2$
fi
```

As we are only concerned with preserving branch-coverage adequate sets of test inputs, it is legitimate to transform this into the following program since a set of test inputs covers all branches of one if and only if it covers all branches of the other.

```
if $P$ then
  $S_2$
else
  $S_1$
fi
```

While the transformations that preserve branch coverage do not need to preserve the functionality of a program, in practice it is useful if they do. This is because transformations that do not preserve the functionality of the transformed section S of a program p must consider the context in which S lies in p. For example, the transformation given above does not need to be valid when applied to a section of a program since it may alter the branches covered after the section is executed. For this reason, the transformation rules introduced in this paper preserve functional equivalence.

While it is helpful for our transformations to preserve functional equivalence, we do not wish to impose this restriction since this eliminates potentially useful transformations. Consider, for example, a statement s of a program p such that for every branch b of p, s does not affect the condition under which b is executed. We can define a transformation that deletes s from p. Then, since this statement does not affect the conditions under which branches are executed, this transformation preserves the sets of branch-coverage adequate test inputs. Where we can eliminate a significant number of statements in this way, such transformations could allow us to produce a program with a shorter execution time. This would help when applying techniques such as genetic algorithms that can require us to execute our program many thousands of times. Thus, there are potentially useful transformations that do not preserve functional equivalence.

As seen in Section 2.3.1, functional equivalence is too weak for our purpose since it allows the branch structure (and even the statement structure) of the original program to be altered dramatically; functional equivalence makes no structural (syntactic) requirements on the transformation process. Unfortunately, as shown in Section 4, previous results from existing work on restructuring transformations show that multiple-level (and even single-level) `exit` statements cannot be removed under path equivalence. This is the motivation for the introduction of branch-coverage equivalence in Section 5.

4. RELATED WORK

Work related to this paper falls into two categories: previous work on transformation to address the problem of poor structure and previous work on program transformation to improve software testability. There is a large amount of work previously done on the former, but comparatively little on the latter. This section first summarizes previous work on testability and transformation to improve testability before moving on to present an overview of related work on restructuring transformations.

Testability has been defined by Voas [28], in terms of the propagation, infection and execution (PIE) framework. The PIE method, a mutation testing-based approach, measures testability in terms of the likelihood that an infection (a fault) is executed and subsequently propagated in an observable way. Voas dealt with testability measurement.

Harman *et al.* [23] introduce testability transformation: the transformation of programs with the goal of improving the ability to test the programs. Testability transformation focuses on ameliorating problems for automated test data generation techniques. The novel aspects of testability transformation theory are discussed in [23], where it is noted that new forms of equivalence are required by testability transformation. In this paper, we introduce an example of a testability transformation that requires such a new form of equivalence and a new proof obligation.

There is a large body of existing literature on the `goto` removal problem [21, 29, 30, 31, 32, 33, 34, 35]. Much of this work considers the problem to be one of removing `goto` statements, replacing them with programs which retain the `exit` (or `break`) statements though they have no `goto` statements.

Some authors consider such `exit` statements to be 'structured' forms of `goto` [17, 36]. For the purpose of test data generation, the aim is to replace all unstructuredness (including `exit` statements) to create single-entry, single-exit constructs. Previous work which retains `exit` statements is not, therefore, directly useful. However, previous work on multiple-level `exit`s is useful because it establishes the expressiveness of these constructs, in particular it shows that:

- multiple-level `exit` statements capture the full expressive power of arbitrary unstructuredness (and that single-level `exit` statements do not);
- `goto` statements cannot always be removed while preserving path equivalence.

The first of these motivates our choice of multiple-level `exit` statements as the paradigm of unstructuredness to be considered, while the second motivates the need for additional transformations which, while not path equivalence preserving, nonetheless preserve aspects of the structure of a program.

4.1. Restructuring transformations

Ramshaw [21] presents an important result concerning multiple-level `exit` statements:

THEOREM 4.1 (RAMSHAW, 1988). *Under path equivalence an arbitrary `goto` program can be transformed into one with no `goto` statements, but which may contain multiple-level `exit` statements.*

This result shows that multiple-level `exit` statements are powerful enough to capture the expressiveness of `goto` statements; no new variables need be introduced and the program's structure (the set of paths it traverses) is preserved. Clearly, path preservation implies branch preservation. It is on this result that we base our claim to consider the full generality of the unstructuredness problem for branch coverage in the present paper.

Previous work has also shown that multiple-level `exit` statements are more powerful than single-level `exit` statements [34].[2]

THEOREM 4.2 (PETERSON, 1973). *Under path equivalence, it is not always possible to transform an arbitrary `goto` program to one with no `goto` statements, but which may contain single-level `exit` statements.*

Knuth, Floyd and Hopcroft show[3] that it is not possible to remove `goto` statements from arbitrary unstructured programs while preserving path equivalence.

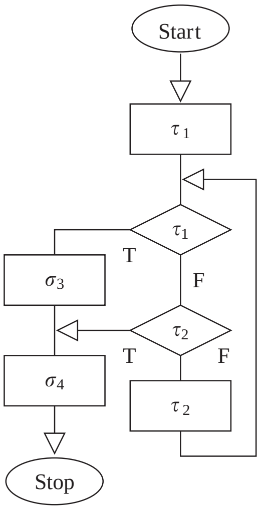


FIGURE 3. Knuth, Floyd and Hopcroft's example.

THEOREM 4.3 (KNUTH, FLOYD AND HOPCROFT, 1971). *There are programs, the `goto`s of which cannot be eliminated under path equivalence.*

Knuth and Floyd use 'regular expression semantics' for flowcharts. The regular expression captures the possible paths through the flowchart, thereby capturing path equivalence. They then define a regular expression class $\mathcal{R}$ which describes paths through programs in a language which is essentially that used in the present paper, but with repeat loops. The addition of repeat loops is not significant. Knuth and Floyd show that the following regular expression cannot be converted into an equivalent one in $\mathcal{R}$:

$$\sigma_1(\tau_{1\mathrm{F}}\tau_{2\mathrm{F}}\sigma_2)^* \ (\tau_{1\mathrm{T}}\sigma_3 \mid \tau_{1\mathrm{F}}\tau_{2\mathrm{T}})\sigma_4$$

The flow chart corresponding to this regular expression is depicted in Figure 3. As can be readily seen, this flowchart is not in some way 'pathological'. The problem `goto` is simply a jump out of a `while` loop that targets a point a little further down (lexically) from the end of the `while` loop.

Since we know that any `goto` program could be converted into a path-equivalent program in which the only unstructuredness is due to `exit` statements, we have the following result.

THEOREM 4.4. *Programs exist whose only unstructuredness is due to `exit` statements that are not path equivalent to any structured program.*

The result provides us with the motivation for a definition of a new notion of equivalence and branch-coverage preserving transformations. While the transformations introduced do not preserve path equivalence, they do preserve branch-coverage adequate sets of test inputs.

5. BRANCH-PRESERVING TRANSFORMATIONS

This section defines what it means for a transformation to preserve branch coverage. It will thus state a requirement

[2] Ramshaw [21] claims that this result is attributed to Kosaraju [33] in 1974. However, it appears that the same result was shown by Peterson in 1973.

[3] In [36], the proof of this result is attributed to J. Hopcroft, although Hopcroft is not an author of the paper.

for the transformation system, based on a new notion of equivalence, that will be developed in Section 6. First, we will introduce some notation.

DEFINITION 5.1. (PROGRAM BRANCHES). *Given a program p, $B(p)$ will denote the branches of p.*

DEFINITION 5.2. (BRANCH COVERAGE CONDITION). *For each branch $b \in B(p)$, there is an associated condition $c(b, p)$ on the input to p such that: test input t leads to b being covered if and only if $c(b, p)(t)$ is $true$. The expression $c(b, p)(t)$ will also be written $c(b, p, t)$.*

We wish to transform a program p into a program p' which has the property that if we generate a set T of test inputs that provides 100% branch coverage for p' then T satisfies 100% branch coverage for p. This relationship between p' and p is captured by the following definition.

DEFINITION 5.3. (BRANCH COVERAGE SUBSUMPTION). *Program p' branch-coverage subsumes p if and only if for every set T of test inputs, if T provides 100% branch coverage for p' then T provides 100% branch coverage for p.*

Naturally, we only need to consider feasible branches.

DEFINITION 5.4. (FEASIBLE BRANCHES). *Branch b of program p is feasible if and only if $\exists t \cdot c(b, p, t)$. The set of feasible branches of p will be denoted $\overline{B}(p)$.*

PROPOSITION 5.1. *Program p' branch-coverage subsumes program p if and only if the following holds:*

$$\forall b \in \overline{B}(p) \cdot \exists b' \in \overline{B}(p') \cdot c(b', p', t) \Rightarrow c(b, p, t)$$

Proof. Case 1: $\Rightarrow$

Suppose p' branch-coverage subsumes program p and $b \in \overline{B}(p)$. Proof by contradiction: suppose that for all $b' \in \overline{B}(p')$ we have that $c(b', p', t) \not\Rightarrow c(b, p, t)$.

Now choose a set of test inputs in the following way. For every $b' \in \overline{B}(p')$ choose a test input $t_{b'}$ such that $c(b', p', t_{b'}) \wedge \neg c(b, p, t_{b'})$. Since $c(b', p', t) \not\Rightarrow c(b, p, t)$, there must be some such $t_{b'}$. Then the resultant set T of test inputs provides 100% branch coverage for p' but does not cover the feasible branch b of p. This contradicts the assumption that p' branch-coverage subsumes program p as required.

Case 2: $\Leftarrow$

Suppose that for all $b \in \overline{B}(p)$ there exists $b' \in \overline{B}(p')$ such that $c(b', p', t) \Rightarrow c(b, p, t)$. Proof by contradiction: suppose that p' does not branch-coverage subsume program p. Then there exists a set T of test inputs that provides 100% branch coverage for p' but does not provide 100% branch coverage for p. Suppose that T does not cover the feasible branch b of p. It is now sufficient to note that $\exists b' \in \overline{B}(p') \cdot c(b', p', t) \Rightarrow c(b, p, t)$ and, since T covers all branches of p', there is some $t_{b'} \in T$ such that $c(b', p', t_{b'})$. Thus $c(b, p, t_{b'})$ and so T must cover b, providing a contradiction as required. □

If p' branch-coverage subsumes p, then if we want a set of test inputs to cover all the feasible branches of p it is sufficient to produce a set of test inputs that covers all feasible branches of p'. However, in producing a set of test inputs for p' we may be achieving more than is required. This observation motivates the following desirable property.

DEFINITION 5.5. (BRANCH COVERAGE EQUIVALENCE). *Program p' is branch-coverage equivalent to p if and only if for every set T of test inputs, T provides 100% branch coverage for p' if and only if T provides 100% branch coverage for p.*

6. A TRANSFORMATION ALGORITHM

This section defines a transformation algorithm that takes a program p which contains one or more `exit` statements and transforms it to a branch-coverage equivalent program p' which does not contain any `exit` statements. The `exit` statements may be multiple-level, but each loop may have only a single `exit` statement which exits the loop. Future work will cover more general transformation algorithms and transformation rules that take a program containing loops with more than one `exit` statement and return a branch-coverage equivalent program in which each loop contains at most one `exit` statement. Consider, for example, the following program.

```
while $P$ do
  $S$
  if $P'$ then
    $S_1$;
    exit $n$;
    $S_2$;
  else
    $S_3$;
    exit $n$;
    $S_4$;
  fi
  $S_5$;
od
```

This is branch-coverage equivalent to the following program that contains only one `exit` statement in its loop.

```
while $P$ do
  $S$
  if $P'$ then
    $S_1$;
  else
    $S_3$;
  fi
  exit $n$;
  $S_5$;
od
```

It will be assumed that all expressions in each predicate of p (of either an `if` statement or a `while` loop) contained in a loop that has an `exit` statement are side-effect free—they cannot alter the value of any program variable. Where a program p contains predicates with side-effects, it may be transformed to form a program p' that does not contain such side-effects [37, 38]. The problem of producing a complete set of branch-coverage equivalence preserving transformations that remove side-effects from predicates will form a significant element of future work.

Section 6.1 outlines a set of transformation rules which are used in Section 6.2 to define a transformation algorithm. Section 6.3 provides a proof of correctness and Section 6.4 proves that the algorithm has low-order polynomial time complexity.

6.1. Transformation rules

In order to simplify the exposition, the only control structures considered in this paper are `while-do-od` loops and `if-then-else-fi` statements. Naturally, `if-then-fi` structures can be dealt with by applying the following branch-coverage equivalence preserving transformation.

RULE 1. *The following may be applied to an* `if-then-fi` *statement.*

```
                  if P then
if P then           S
  S          →    else
fi                  skip
                  fi
```

We argue that other standard control constructs may be dealt with using rules similar to those described here.

At times we will want to talk about a subprogram of a program p. This will be defined in the following way.

DEFINITION 6.1. (SUBPROGRAM). *The concept of one program being a subprogram of another is defined by the following rules.*

(*i*) *S is a subprogram of S.*
(*ii*) *Given any program p,* `skip` *is a subprogram of p.*
(*iii*) *S is a subprogram of* S`;`S'.
(*iv*) *S is a subprogram of* S'`;`S.
(*v*) *S_1 is a subprogram of* `if` P `then` S_1 `else` S_2 `fi`.
(*vi*) *S_2 is a subprogram of* `if` P `then` S_1 `else` S_2 `fi`.
(*vii*) *S is a subprogram of* `while` P `do` S `od`.
(*viii*) *If p' is a subprogram of p and p'' is a subprogram of p' then p'' is a subprogram of p.*

We say that S_1 is contained in S_2 if S_1 is a subprogram of S_2. Each `exit` statement is contained within the body of a loop. At points, when considering a statement s we will want to talk about the body of the innermost loop that contains s.

DEFINITION 6.2. (INNERMOST LOOP BODY). *Given* `exit` *statement s in program p, $subprog(s, p)$ is the maximum subprogram, S, of p such that s is contained in S and s is not contained within a loop in S.*

Where a statement s lies within the body of the loop `while` P `do` S `od`, and is not contained in a loop within S, it is possible to define the depth, $depth(s, S)$ of s in this loop. The term $depth(s, S)$ will denote how far s is nested, inside `if` statements, within S.

DEFINITION 6.3. (DEPTH). *The function depth is defined by the following rules.*

$$depth(s, s) = 0$$

$$depth(s, S;S') = \begin{cases} depth(s, S) \\ \quad \text{if } s \text{ is in } S \\ depth(s, S') \\ \quad \text{otherwise} \end{cases}$$

$$depth\left(s, \begin{array}{l} \texttt{if } P \\ \texttt{then } S \\ \texttt{else } S' \\ \texttt{fi} \end{array} \right) = \begin{cases} 1 + depth(s, S) \\ \quad \text{if } s \text{ is in } S \\ 1 + depth(s, S') \\ \quad \text{otherwise} \end{cases}$$

The precondition for $depth(s, S)$ is that s is an `exit` *statement contained within a loop-free section of S.*

The above definition assumes that `exit` statements are uniquely identified: where there are two syntactically equivalent `exit` statements, we label these in order to distinguish them.

DEFINITION 6.4. (DEPTH OF AN EXIT). *Given* `exit` *statement s in program p, the depth of s in p is $depth(s, S)$, where $S = subprog(s, p)$.*

We now introduce transformations rules that will be used to eliminate (possibly multiple-level) `exit` statements while preserving branch-coverage equivalence. In order to do this, we examine the following cases:

(i) an `exit` statement of depth 0;
(ii) an `exit` statement of depth 1;
(iii) an `exit` statement of depth >1.

Transformation steps for each of these cases are given with the assumption that an `exit` statement s is represented as `exit` n, where n denotes the number of levels over which the `exit` operates. Essentially, the algorithm takes `exit` statements and repeatedly transforms the program in order to reduce their level. Once the level of an `exit` statement has been reduced to 0 it is replaced by `skip`.

A single-level `exit` will be represented by `exit` 1. It transpires that the transformation rules may reduce a single-level `exit` to the statement `exit` 0. We thus introduce the following transformation that cleans up any such terms generated in the transformation process.

RULE 2.

```
exit 0  →  skip
```

The following are immediate.

PROPOSITION 6.1. *If p may be transformed into p' using an application of Rule 2 then p' is branch-coverage equivalent to p.*

PROPOSITION 6.2. *If p may be transformed into p' using an application of Rule 2 then p' and p are functionally equivalent.*

While functional equivalence is not required for branch-coverage equivalence, it is useful to have functional equivalence preserving transformations since these cannot affect the behaviour of the program on any code executed after the transformed fragment, making the application of the transformations context independent.

RULE 3. *The following rule may be applied to an* `exit` *statement of depth* 0.

```
                      if $P$ then
while $P$ do            $S$;
  $S$;                  exit $n-1$
  exit $n$;      →    else
  $S'$                  skip
od                    fi
```

This has the precondition that $n > 0$.

PROPOSITION 6.3. *If* p *may be transformed into* p' *using an application of Rule 3 then* p' *is branch-coverage equivalent to* p.

Proof. Suppose that the rule is applied to a subprogram X_1 of p of the form

```
while $P$ do
  $S$;
  exit $n$;
  $S'$
od
```

to create a subprogram X_2 of p' of the form

```
if $P$ then
  $S$;
  exit $n-1$
else
  skip
fi
```

Clearly, paths in p that do not pass through X_1 are not affected by this transformation. Consider a feasible path π through p that passes through X_1. Here, it is important to observe that the loop may iterate at most once.

Each time X_1 is met on π, there are two cases to consider:

(i) P is true in the state before X_1 is executed. In this case the path passes through S and then executes `exit` n.
(ii) P is false in the state before X_1 is executed. The path π then exits this section of the code.

Now consider X_2. Each time X_2 is met in a path, there are two cases to consider:

(i) P is true in the state before X_2 is executed. In this case the path passes through S and then executes `exit` $n-1$.
(ii) P is false in the state before X_2 is executed. The path then exits this section of the code.

The statement `exit` n, in X_1, is equivalent to the statement `exit` $n-1$ in X_2. Thus, X_1 and X_2 are functionally equivalent. In addition, the branches in X_1 and X_2 are followed under the same conditions. The result thus follows. □

The following may be proved in a similar manner.

PROPOSITION 6.4. *If* p *may be transformed into* p' *using an application of Rule 3 then* p' *and* p *are functionally equivalent.*

RULE 4. *The following may be applied to an* `exit` *statement of depth* 1 *within the* `then` *part of an* `if` *statement.*

```
while $P$ do
  $S$;               while ($P \wedge \neg P''$) do
  if $P'$ then         $S$;
    $S_1$;             $S_3$;
    exit $n$;          $S_4$
    $S_2$            od;
  else          →    if $P$ then
    $S_3$              $S$;
  fi                   $S_1$;
  $S_4$                exit $n-1$
od                   fi
```

Here P'' *denotes the result of evaluating* P' *after executing* S. P'' *is described in greater detail below. This rule has the precondition that* $n > 0$.

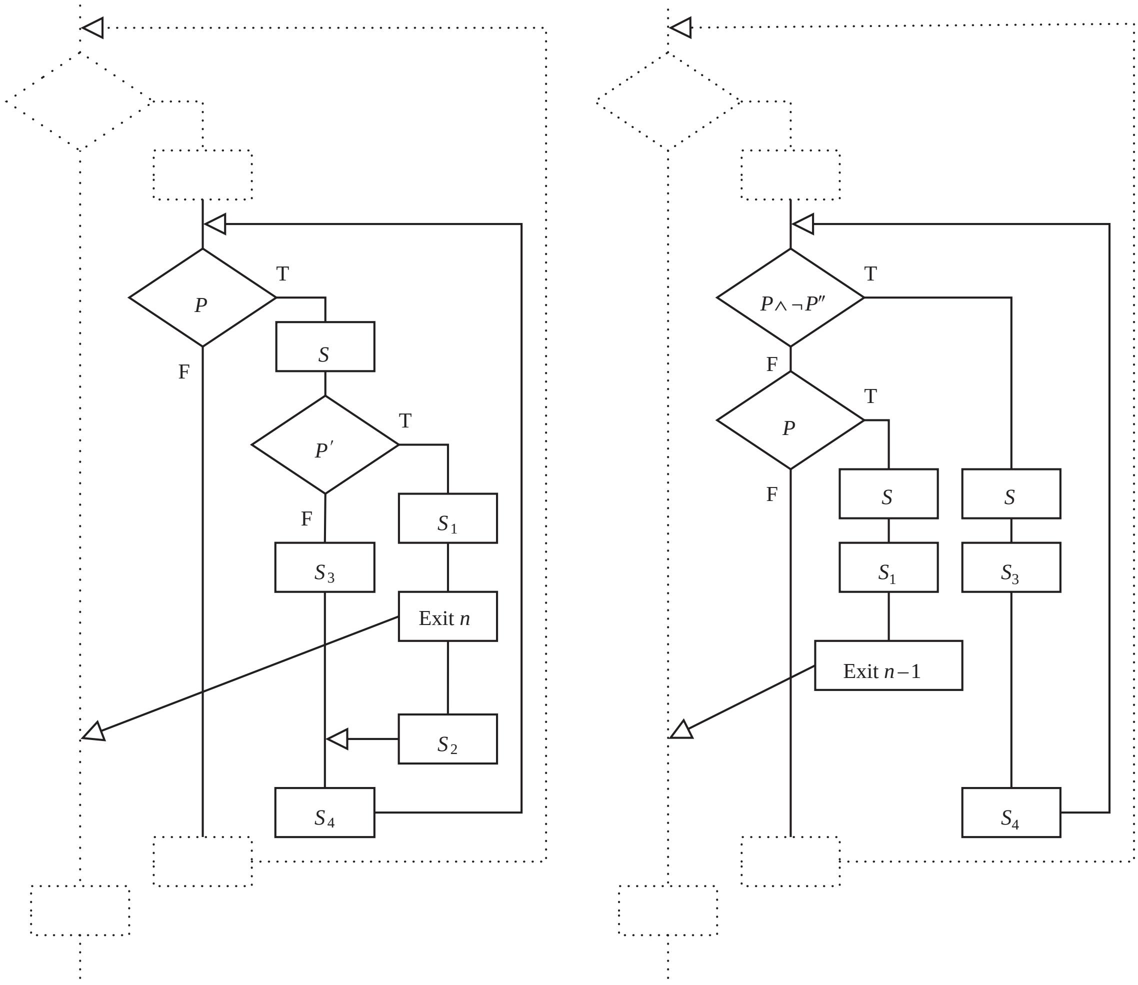
Rule 4 is illustrated in Figure 4.

The transformation which produces P'' from S and P' consists of making P'' a call to a new function ϕ. The local variables of ϕ are the variables of S which are defined before they are used [39]. The formal (value) parameters of ϕ are the variables of S which are not defined before use (i.e. those used before they are defined or those simply used and not defined at all).

The function ϕ is a predicate; it returns a Boolean type result. The body of ϕ consists of the statement block S, followed by a return statement. The return statement simply returns the result of evaluating the expression P'.

Observe that P' is guaranteed to have the same meaning in the context of the return statement from ϕ as it does from the original point in the program in which it occurs. Any variable (mentioned in P') which is defined before it is used, can be replaced by a local variable; since its value is defined before use it does not lose any previous value. Any variable (mentioned in P') which is used before it is defined will be passed (by value) to ϕ, thereby creating a local copy. Any variable simply used and not defined will be passed as a formal parameter. Clearly, a variable must be defined either before it is used or not and so all variables mentioned in P' will be available (with their correct value) at the point of the return statement.

Also, it is observed that the call to the function ϕ has no side-effects, since all variables which are defined by the copy



Before

After

FIGURE 4. Illustration of Rule 4, showing the reduction in depth of the `exit` statement.

of S in the body of ϕ are either local or are formal value parameters. These transformations assume that the statement S performs no input/output, or any other implicit [40] state update and that the expression P' is side-effect free.

PROPOSITION 6.5. *If p may be transformed into p' using an application of Rule 4 then p' is branch-coverage equivalent to p.*

Proof. Suppose that the rule is applied to a subprogram X_1 of p, of the form

```
while $P$ do
  $S$;
  if $P'$ then
    $S_1$;
    exit $n$;
    $S_2$
  else
    $S_3$
  fi
  $S_4$
od
```

to create a subprogram X_2 of p' of the form

```
while ($P \wedge \neg P''$) do
  $S$;
  $S_3$;
  $S_4$
od;
if $P$ then
  $S$;
  $S_1$;
  exit $n-1$
fi
```

Clearly, paths in p that do not pass through X_1 are not affected by this transformation. Consider a path π of p that passes through X_1. Consider a point at which the path π passes through X_1.

Observe that the conditions for leaving the loops in X_1 and X_2 are equivalent.

There are now two cases to consider for X_1:

(i) The loop terminates due to P being false, or becoming false. Thus, the program passes through S; S_3; S_4 zero or more times until P is false.

(ii) The loop terminates due to P' becoming true. Thus, the program repeatedly passes through S; S_3; S_4, then S followed by P' being true. The program then passes through S_1 before meeting `exit` n.

Now consider X_2. Again there are two cases.

(i) The loop is left due to P being false, or becoming false. Thus, the program passes through S; S_3; S_4 zero or more times until P is false.
(ii) The loop is left due to $\neg P''$ becoming false. In this case, upon leaving the loop P must be true and thus this is followed by S; S_1;`exit` $n - 1$. Thus, the program repeatedly passes through S; S_3; S_4, then S followed by P' being true. The program then passes through S_1 and meets `exit` $n - 1$.

The branches contained in P'' are equivalent to those in S and are executed under the same condition as S in X_2. Thus, we may ignore the branches in P'' when determining whether 100% branch coverage has been achieved for p'.

The result now follows by observing that the statement `exit` n in X_1 is equivalent to the statement `exit` $n - 1$ in X_2 and the equivalence of the cases for X_1 and X_2; these correspond to branches. □

A similar argument may be used to prove the following.

PROPOSITION 6.6. *If p may be transformed into p′ using an application of Rule 4 then p′ is functionally equivalent to p.*

The following is equivalent to Rule 4, except with the `exit` statement in the `else` part of the `if` statement.

RULE 5. *The following may be applied to an* `exit` *statement of depth* 1 *within the* `else` *part of an* `if` *statement.*

```
while $P$ do
  $S$;
  if $P'$ then
    $S_1$;
  else
    $S_2$;
    exit $n$;
    $S_3$
  fi
  $S_4$
od
```

→

```
while ($P \wedge P''$) do
  $S$;
  $S_1$;
  $S_4$
od;
if $P$ then
  $S$;
  $S_2$;
  exit $n-1$
fi
```

Here P'' denotes the result of evaluating P' after executing S. This rule has the precondition that $n > 0$.

The proofs of the following are equivalent to those of Propositions 6.5 and 6.6.

PROPOSITION 6.7. *If p may be transformed into p′ using an application of Rule 5 then p′ is branch-coverage equivalent to p.*

PROPOSITION 6.8. *If p may be transformed into p′ using an application of Rule 5 then p′ is functionally equivalent to p.*

Where an `exit` statement s has depth >1 we may use Rule 6, below, in order to reduce its depth. Thus, repeated application of Rule 6 may be used to reduce the depth of s to 1 whereupon Rule 4 or 5 may be applied.

RULE 6. *The following may be applied to an* `exit` *statement of depth* >1

```
if $P$ then
  $S_1$;
  if $P'$ then
    $T_1$;
    exit $n$;
    $T_2$
  else
    $T_3$;
  fi
  $S_2$
else
  $S_3$
fi
```

→

```
if ($P \wedge P''$) then
  $S_1$;
  $T_1$;
  exit $n$
else
  if $P$ then
    $S_1$;
    $T_3$;
    $S_2$
  else
    $S_3$
  fi
fi
```

P'' is the result of evaluating P' after executing S_1. This rule has the precondition that $n > 0$.

Rule 6 is illustrated in Figure 5.

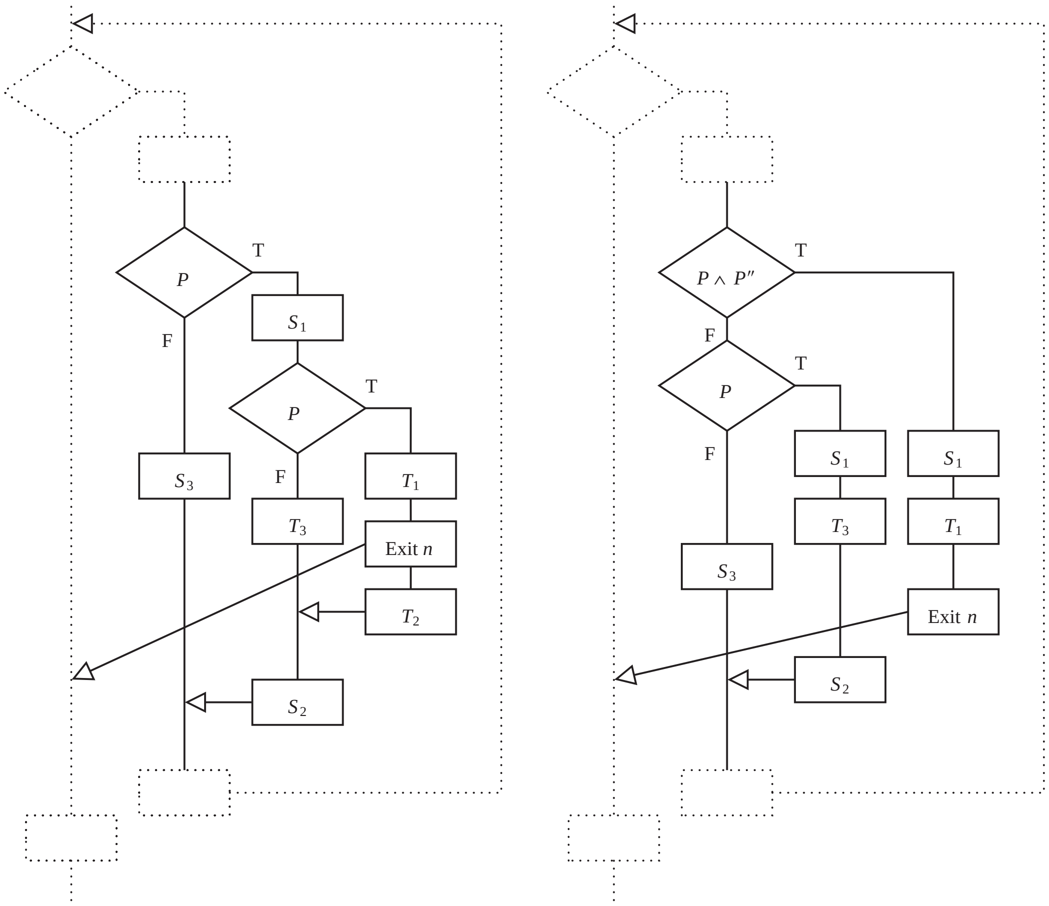
PROPOSITION 6.9. *If p may be transformed into p′ using an application of Rule 6 then p′ is branch-coverage equivalent to p.*

Proof. Suppose that the rule is applied to a subprogram X_1 of p, of the form

```
if $P$ then
  $S_1$;
  if $P'$ then
    $T_1$;
    exit $n$;
    $T_2$
  else
    $T_3$
  fi
  $S_2$
else
  $S_3$
fi
```

to create a subprogram X_2 of p' of the form

```
if ($P \wedge P''$) then
  $S_1$;
  $T_1$;
  exit $n$
else
  if $P$ then
    $S_1$;
    $T_3$;
    $S_2$
  else
    $S_3$
  fi
fi
```



Before

After

FIGURE 5. Illustration of Rule 6.

Paths in p that do not pass through X_1 are not affected by this transformation. Consider a path π of p that passes through X_1. Consider a point at which π passes through X_1. There are three cases.

(i) P is false and S_3 is executed.
(ii) P is initially true and P' is true when evaluated after S_1. Here the program passes through S_1; T_1; and then `exit` n.
(iii) P is initially true and P' is false when evaluated after S_1. Here the program passes through S_1; T_3; S_2.

Now consider X_2. Again there are three cases.

(i) $P \wedge P''$ is initially false and P is initially false. Here S_3 is executed.
(ii) $P \wedge P''$ is initially true. Here the program passes through S_1; T_1; and then `exit` n.
(iii) $P \wedge P''$ is initially false and P is initially true. Here the program passes through S_1; T_3; S_2.

The branches in P'' are equivalent to those in S_1 in X_2 and are executed under the same conditions (P being true).

The result follows by observing that the three cases for X_2 are equivalent to the three cases for X_1 and that in each case the conditions defined by the cases correspond to the branches. □

PROPOSITION 6.10. *If p may be transformed into p' using an application of Rule 6 then p' is functionally equivalent to p.*

The transformation rule only deals with `exit` statements in the `then` case of an `if` statement. We thus get three additional rules. These will now be stated.

RULE 7. *The following may be applied to an* `exit` *statement of depth* >1.

```
if $P$ then
  $S_1$;
  if $P'$ then
    $T_1$;
  else
    $T_2$;
    exit $n$;
    $T_3$
  fi
  $S_2$
else
  $S_3$
fi
```

→

```
if ($P \wedge \neg P''$) then
  $S_1$;
  $T_2$;
  exit $n$
else
  if $P$ then
    $S_1$;
    $T_1$;
    $S_2$
  else
    $S_3$
  fi
fi
```

P'' is the result of evaluating P' after executing S_1. This rule has the precondition that $n > 0$.

RULE 8. *The following may be applied to an* `exit` *statement of depth* >1.

```
if $P$ then
  $S_1$
else
  $S_2$;
  if $P'$ then
    $T_1$;
    exit $n$;
    $T_2$
  else
    $T_3$
  fi
  $S_3$
fi
```

→

```
if $(\neg P \wedge P'')$ then
  $S_2$;
  $T_1$;
  exit $n$
else
  if $\neg P$ then
    $S_2$;
    $T_3$;
    $S_3$
  else
    $S_1$
  fi
fi
```

P'' is the result of evaluating P' after executing S_2. This rule has the precondition that $n > 0$.

RULE 9. *The following may be applied to an* `exit` *statement of depth* >1.

```
if $P$ then
  $S_1$
else
  $S_2$;
  if $P'$ then
    $T_1$
  else
    $T_2$;
    exit $n$;
    $T_3$
  fi
  $S_3$
fi
```

→

```
if $(\neg P \wedge \neg P'')$ then
  $S_2$;
  $T_2$;
  exit $n$
else
  if $\neg P$ then
    $S_2$;
    $T_1$;
    $S_3$
  else
    $S_1$
  fi
fi
```

P'' is the result of evaluating P' after executing S_2. This rule has the precondition that $n > 0$.

The proofs of the following are equivalent to those of Propositions 6.9 and 6.10.

PROPOSITION 6.11. *If p may be transformed into p' using an application of Rules 7, 8 or 9 then p' is branch-coverage equivalent to p.*

PROPOSITION 6.12. *If p may be transformed into p' using an application of Rules 7, 8 or 9 then p' is functionally equivalent to p.*

6.2. Transformation algorithm

This section describes a transformation algorithm based on the rules defined in Section 6.1. The algorithm is structured in order to guarantee that each rule's precondition holds when it is applied. Section 6.3 proves that this algorithm is correct and Section 6.4 explores its algorithmic complexity.

The function *tdepth*, which gives the nesting level of a statement in a program, will be used in the algorithm in order to ensure that the most deeply nested `exit` statements are transformed out first.

DEFINITION 6.5. (TDEPTH). *The function tdepth is defined by the following rules.*

$$tdepth(s, s) = 0$$

$$tdepth(s, S; S') = \begin{cases} tdepth(s, S) & \text{if } s \text{ is in } S \\ tdepth(s, S') & \text{otherwise} \end{cases}$$

$$tdepth\left(s, \begin{array}{l} \texttt{if } P \\ \texttt{then } S \\ \texttt{else } S' \\ \texttt{fi} \end{array}\right) = \begin{cases} 1 + tdepth(s, S) & \text{if } s \text{ is in } S \\ 1 + tdepth(s, S') & \text{otherwise} \end{cases}$$

$$tdepth\left(s, \begin{array}{l} \texttt{while } P \texttt{ do} \\ S \\ \texttt{od} \end{array}\right) = 1 + tdepth(s, S)$$

The precondition for $tdepth(s, S)$ is that s is a statement contained within S.

The following is the transformation algorithm.

(i) Input program p.
(ii) While p contains one or more `exit` statements do:
(iii) Choose an `exit` statement s of p with greatest *tdepth*.
(iv) Let n denote the depth of s in p.
(v) If $n = 0$ then apply Rule 3 to s.
(vi) If $n = 1$ then apply Rule 4 or 5 to s.
(vii) If $n > 1$ then apply one of Rules 6, 7, 8 or 9 to s.
(viii) Repeatedly apply Rule 2 until there are no instances of `exit` 0 in p.
(ix) od
(x) Output p.

6.3. Proof of correctness

We first prove a property of the algorithm that will be used to prove that it must terminate.

LEMMA 6.1. *Suppose that s is an* `exit` *statement in program S and that an application of one of Rules 6, 7, 8 or 9 in the transformation algorithm to an* `exit` *statement $s' \neq s$, transforms S into S'. Then $tdepth(s, S) \geq tdepth(s, S')$.*

Proof. Assume that the rule being applied is Rule 6—the other cases follow by a similar argument. Proof by contradiction: suppose that $tdepth(s, S) < tdepth(s, S')$. Recall that Rule 6 applied to `exit` statement s' is:

```
if $P$ then
  $S_1$;
  if $P'$ then
    $T_1$;
    $s'$;
    $T_2$
  else
    $T_3$
  fi
  $S_2$
else
  $S_3$
fi
```

→

```
if $(P \wedge P'')$ then
  $S_1$;
  $T_1$;
  $s''$
else
  if $P$ then
    $S_1$;
    $T_3$;
    $S_2$
  else
    $S_3$
  fi
fi
```

If s' is `exit` n then s'' is the `exit` statement `exit` $n-1$.

Clearly, this transformation can only increase the value of *tdepth* for s if s is contained in S_1. But if s is contained in S_1 then, since each `while` loop has at most one `exit` statement associated with it, $tdepth(s, S) > tdepth(s', S)$. This contradicts the algorithm applied to s' in S. □

PROPOSITION 6.13. *The transformation algorithm is guaranteed to terminate.*

Proof. We need to consider only the steps in the loop. It is sufficient to observe that:

(i) Each iteration converts a level n `exit` into a level $n-1$ `exit` or reduces the value of *tdepth* for an `exit`.
(ii) Any instance of `exit 0` generated by this process is removed.
(iii) Since the algorithm applies a rule to an `exit` statement of maximum $tdepth$, no step may introduce a new `exit`.
(iv) No step may convert a level n `exit` into a level m `exit` for some $m > n$.
(v) By Lemma 6.1, no step can increase the value of $tdepth$ of an `exit`.

Thus each iteration of the loop reduces the sum of the levels of the `exit` statements plus the sum of *tdepth* for each `exit`. The result follows from observing that this value is an integer and is bounded below by 0. □

PROPOSITION 6.14. *When applied to a program p, the transformation algorithm always terminates with a structured program p'.*

Proof. This follows immediately from the observation that the only constructs that lead to unstructured programs are `exit` statements and the program cannot terminate if one or more `exit` statements remain. □

THEOREM 6.1. *The transformation algorithm is correct.*

Proof. By Proposition 6.13, the algorithm must terminate. From Propositions 6.3, 6.5, 6.7, 6.9 and 6.11, we know that the transformation rules preserve branch-coverage equivalence. By Proposition 6.14 we know that the algorithm terminates with a structured program.

Thus, the algorithm must terminate and the result must be a structured program that is branch-coverage equivalent to the original. Thus the algorithm is correct. □

By taking a program p and returning a structured branch-coverage equivalent program p' we simplify the problem of generating a set of test inputs that satisfies 100% branch coverage since we can now apply a range of test generation algorithms to p'. Naturally, having generated a set T of test inputs from p' we test p with the elements of T.

6.4. Algorithmic complexity

This section proves that the transformation algorithm has low-order polynomial complexity.

THEOREM 6.2. *Suppose p has `exit` statements $s_1, \ldots, s_n$, `exit` statement s_i is an n_i level `exit` statement and let $d_i = tdepth(s_i, p)$. Let $d = \sum_{i=1}^{n} d_i$ and $m = \sum_{i=1}^{n} n_i$. The transformation algorithm has time complexity of $O(m + d)$.*

Proof. Given `exit` statement s_i, by Lemma 6.1, Rules 6, 7, 8 and 9 are applied at most d_i times to s_i. Thus Rules 6, 7, 8 and 9 are applied at most d times in total.

Rules 3, 4 and 5 reduce the level of one `exit` and do not increase the level of any other `exit`. Since a rule is applied to an `exit` statement of maximum *tdepth* and each loop has at most one `exit` statement, no step can create a new `exit` statement. Thus, the number of applications of these three rules is bounded above by m.

To conclude, the number of applications of Rules 3, 4 and 5 is bounded above by m and the number of applications of Rules 6, 7, 8 and 9 is bounded above by d. Clearly, the number of applications of Rule 2, which cleans up terms of the form `exit 0`, is also bounded above by m. The result thus follows. □

It is to be noted that where we have several procedures or functions, the algorithm may be separately applied to these. Thus, the complexity is linear in the number procedures and functions.

7. CONCLUSIONS

Many test data generation techniques are based upon white-box test criteria. Tools and algorithms that automatically generate test data in order to satisfy a white-box test criterion are hindered by unstructured control flow. The need for a restructuring transformation to preserve the test adequacy criterion makes this problem different from the traditional, well-known problem of program restructuring transformation.

This paper uses a novel transformation approach in which the program is restructured in such a way that branch-coverage adequate sets of test inputs are preserved. This form of transformation does not necessarily need to preserve traditional (functional) equivalence, as the transformed program is only required to generate test data. It does, however, need to preserve the sets of branch-coverage adequate test inputs. This new preservation constraint has important theoretical and practical implications and entails proof obligations in terms of branch-coverage adequate sets of test inputs, rather than, for example, functional equivalence.

The approach is illustrated with a set of transformation rules for multiple-level `exit` statement removal and a simple transformation algorithm for restructuring which uses these rules. The algorithm is proved correct insofar as the requirement that it preserves the branch-coverage adequate sets of test inputs.

Future work will address the problem of determining, where possible, branch-coverage equivalence. While it is clear that, in general, branch-coverage equivalence is not decidable, there may exist useful conservative decision procedures. The existence of such procedures would simplify the process of verifying transformation rules and might have the potential to assist in the automated generation of such rules. We also intend to investigate alternative white-box test criteria such as 100% statement coverage and 100% MC/DC coverage [41]. We propose that notions similar to branch-coverage equivalence will prove useful for many such test criteria.

The transformation algorithm given in this paper was designed to be applied to programs whose predicates (contained in loops that have an `exit` statement) are side-effect free and in which each loop contains at most one `exit` statement. Thus, there remains the problem of finding either a more general transformation algorithm or transformation rules that preserve branch-coverage equivalence and transform a program into one that satisfies these conditions. The novel semantics preserved by branch-coverage preserving transformations suggests the possibility of using slicing to remove parts of the program irrelevant to the satisfaction of a test goal. The incorporation of slicing into our transformation approach is also a topic for future work.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their many helpful comments. In particular, the example used to illustrate the restriction, that we have at most one `exit` statement for a loop, was suggested by one of the reviewers. This work was partially funded by EPSRC grants GR/R43150 (Formal Methods and Testing), GR/R98938 (Testability Transformation) and GR/M78083 (Software Engineering using Metaheuristic INnovative ALgorithms). The work is also supported by two development grants from the DaimlerChrysler Test-Data Generation group, DaimlerChrysler AG Berlin.

REFERENCES

[1] Boehm, B. W. (1981) *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

[2] Clarke, L. A. (1976) A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, **2**, 215–222.

[3] Goldberg, A., Wang, T. C. and Zimmerman, D. (1994) Applications of feasible path analysis to program testing. In Ostrand, T. (ed.) *Proc. 1994 Int. Symp. on Software Testing and Analysis (ISSTA)*, Seattle, WA, August 17–19, pp. 80–94. ACM Press, New York.

[4] King, J. C. (1976) Symbolic execution and program testing. *Commun. ACM*, **19**, 385–394.

[5] DeMillo, R. A. and Offutt, A. J. (1993) Experimental results from an automatic test generator. *ACM Trans. Softw. Eng. Meth.*, **2**, 109–127.

[6] Offutt, A. J. (1990) An integrated system for automatically generating test data. In Ng, P. A., Ramamoorthy, C. V., Seifert, L. C. and Yeh, R. T. (eds) *Proc. First Int. Conf. on Systems Integration*, Morristown, NJ, April 23–26, pp. 694–701. IEEE Computer Society Press, Los Alamitos, California, USA.

[7] Ferguson, R. and Korel, B. (1996) The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Meth.*, **5**, 63–86.

[8] Jones, B., Sthamer, H.-H. and Eyres, D. (1996) Automatic structural testing using genetic algorithms. *Softw. Eng. J.*, **11**, 299–306.

[9] Michael, C., McGraw, G. and Schatz, M. (2001) Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, **27**, 1085–1110.

[10] Mueller, F. and Wegener, J. (1998) A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, Denver, CO June 3–5, pp. 144–154. IEEE Computer Society Press, Los Alamitos, CA, USA.

[11] Pargas, R. P., Harrold, M. J. and Peck, R. R. (1999) Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, **9**, 263–282.

[12] Pohlheim, H. and Wegener, J. (1999) Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E. (eds) *Proc. Genetic and Evolutionary Computation Conference*, Orlando, FL, July 13–17, p. 1795. Morgan Kaufmann, San Francisco, CA 94104, USA.

[13] Schultz, A., Grefenstette, J. and Jong, K. (1993) Test and evaluation by genetic algorithms. *IEEE Expert*, **8**, 9–14.

[14] Tracey, N., Clark, J., Mander, K. and McDermid, J. (2000) Automated test-data generation for exception conditions. *Software Prac. Exper.*, **30**, 61–79.

[15] DeMillo, R. and Offutt, A. J. (1991) Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, **17**, 900–910.

[16] Ferrante, J., Ottenstein, K. J. and Warren, J. D. (1987) The program dependence graph and its use in optimization. *ACM Trans. Progr. Lang. Sys.*, **3**, 319–349.

[17] Agrawal, H. (1994) On slicing programs with jump statements. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, Orlando, FL, June 20–24, pp. 302–312. ACM Press, New York, NY.

[18] Ball, T. and Horwitz, S. (1993) Slicing programs with arbitrary control–flow. In Fritzson, P. (ed.) 1^{st} *Conf. on Automated Algorithmic Debugging*, Linköping, Sweden, May 3–5, *LNCS* **749**, 206–222. Springer-Verlag, Heidelberg.

[19] Choi, J. and Ferrante, J. (1994) Static slicing in the presence of goto statements. *ACM Trans. Progr. Lang. Sys.*, **16**, 1097–1113.

[20] Sinha, S., Harrold, M. J. and Rothermel, G. (1999) System-dependence-graph-based slicing of programs with arbitrary interprocedural control-flow. In *Proc. 21st Int. Conf. on Software Engineering*, Los Angeles, CA, May 16–22, pp. 432–441. ACM Press, New York, NY.

[21] Ramshaw, L. (1988) Eliminating goto's while preserving program structure. *J. ACM*, **35**, 893–920.

[22] Erosa, A. M. and Hendren, L. J. (1994) Taming control flow: a structured approach to eliminating GOTO statements. In *Proc. IEEE Int. Conf. on Computer Languages*, Toulouse, France, May 16–19, pp. 229–240. IEEE Computer Society Press, Los Alamitos, CA.

[23] Harman, M., Hu, L., Hierons, R. M., Wegener, J., Sthamer, H., Baresel, A. and Roper, M. (2004) Testability transformation. *IEEE Trans. Software Eng.*, **30**, 3–16.

[24] Ward, M. (1999) Assembler to C migration using the FermaT transformation system. In *Proc. IEEE Int. Conf. on Software Maintenance (ICSM'99)*, Oxford, UK, August 30–September 3, pp. 67–76. IEEE Computer Society Press, Los Alamitos, CA.

[25] Jones, B. F., Eyres, D. E. and Sthamer, H.-H. (1998) A strategy for using genetic algorithms to automate branch and fault-based testing. *Comp. J.*, **41**, 98–107.

[26] Tracey, N., Clark, J. and Mander, K. (1998) Automated program flaw finding using simulated annealing. In *Proc. Int. Symp. on Software Testing and Analysis*, Clearwater Beach, FL, March 2–5, pp. 73–81. ACM Press, New York, NY.

[27] Wegener, J., Sthamer, H., Jones, B. F. and Eyres, D. E. (1997) Testing real-time systems using genetic algorithms. *Software Qual. J.*, **6**, 127–135.

[28] Voas, J. M. and Miller, K. W. (1995) Software testability: The new verification. *IEEE Software*, **12**, 17–28.

[29] Böhm, C. and Jacopini, G. (1966) Flow diagrams, Turing machines, and languages with only two formation rules. *Commun. ACM*, **9**, 366–372. Presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory.

[30] Cooper, D. (1967) Böhm and Jacopini's reduction of flow charts. *Commun. ACM*, **10**, 463–473.

[31] Dijkstra, E. W. (1972) *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ.

[32] Harel, D. (1980) On folk theorems. *Commun. ACM*, **23**, 379–389.

[33] Kosaraju, S. R. (1974) Analysis of structured programs. *J. Comput. Syst. Sci.*, **9**, 232–255.

[34] Peterson, W. W., Kasami, T. and Tokura, N. (1973) On the capabilities of while, repeat, and exit statements. *Commun. ACM*, **16**, 503–512.

[35] Ward, M. (1994) Reverse engineering through formal transformation. *Comp. J.*, **37**, 795–813.

[36] Knuth, D. E. and Floyd, R. W. (1971) Notes on avoiding 'go to' statements. *Inform. Process. Lett.*, **1**, 23–31.

[37] Dolado, J. J., Harman, M., Otero, M. C. and Hu, L. (2003) An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Softw. Eng.*, **29**, 665–670.

[38] Harman, M., Hu, L., Zhang, X. and Munro, M. (2001) Side-effect removal transformation. In *Proc.* 9^{th} *IEEE Int. Workshop on Program Comprehension (IWPC'01)*, Toronto, Canada, May 12–13, pp. 310–319. IEEE Computer Society Press, Los Alamitos, CA.

[39] Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, MA.

[40] Sivagurunathan, Y., Harman, M. and Danicic, S. (1997) Slicing, I/O and the implicit state. In Kamkar, M. (ed.) 3^{rd} *Int. Workshop on Automated Debugging (AADEBUG'97)*, Linköping, Sweden, May 26–27, Linköping Electronic Articles in Computer and Information Science, **2**, 59–65.

[41] RTCA/DO-178B (1992) *Software Considerations in Airborne Systems and Equipment Certification*. RTA, Washington, DC.