Using Spanning Sets for Coverage Testing

Martina Marré and Antonia Bertolino

Abstract—A test coverage criterion defines a set E_c of entities of the program flowgraph and requires that every entity in this set is covered under some test case. Coverage criteria are also used to measure the adequacy of the executed test cases. In this paper, we introduce the notion of spanning sets of entities for coverage testing. A spanning set is a minimum subset of E_c, such that a test suite covering the entities in this subset is guaranteed to cover every entity in E_c. When the coverage of an entity always guarantees the coverage of another entity, the former is said to subsume the latter. Based on the subsumption relation between entities, we provide a generic algorithm to find spanning sets for control flow and data flow-based test coverage criteria. We suggest several useful applications of spanning sets: They help reduce and estimate the number of test cases needed to satisfy coverage criteria. We also empirically investigate how the use of spanning sets affects the fault detection effectiveness.

Index Terms—Control flow, coverage criteria, data flow, ddgraph, spanning sets, subsumption.

1 Introduction

Various approaches to testing exist. The "classical" way to facilitate testing (as opposed to the more recent test-driven design [4]) is to establish a collection of requirements to be fulfilled, which defines a *test criterion*. In particular, structural coverage criteria map these requirements onto a set of entities in the program flowgraph that must be covered when the test cases are executed. These entities may be derived from the program's control flow or from the program's data flow.

We introduce the new concept of *spanning sets* of entities for a coverage criterion. A spanning set is a *minimum* subset of entities with the property that any set of test cases covering this subset covers every entity in the program. We discuss how this concept can be exploited to make coverage testing more efficient in many respects. We underscore, however, that we are not proposing a new testing method as an alternative to existing ones, but rather an approach to improve the way existing test-coverage criteria are applied.

We have been studying the use of spanning sets of entities in coverage testing for some time. In [6], [7], [9], we identify spanning sets of entities for the *all-branches* criterion and present some useful applications. In [19], we identify spanning sets of entities for the *all-uses* criterion. In this paper, we provide a general method for identifying a spanning set of entities for an entire family of test coverage criteria and discuss its applications.[1] The focus is on unit testing; the application of the approach at the inter-procedural level is an important extension, but is out of the scope of this paper.

1. A concise, preliminary version of this paper appeared in [20].

- M. Marré is with the Departamento de Computacion, FCEyN, Universidad de Buenos Aires, Argentina. E-mail: mmarre@pragma.com.ar.
- A. Bertolino is with the Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo," Area della Ricerca CNR di Pisa, 56100 Pisa, Italy. E-mail: antonia.bertolino@isti.cnr.it.

Manuscript received 18 Nov. 1998; revised 25 Feb. 2003; accepted 10 Apr. 2003.
Recommended for acceptance by M.J. Harrold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 108263.

To the best of our knowledge, there has been no previous work on identifying a minimum set of entities that guarantees full coverage for an entire family of criteria. Some authors [11], [1] have independently recognized this idea for the simple strategy of all-branches coverage, but failed to generalize it to other criteria.

Some work has been done to reduce the size of a test suite. In particular, Gupta and Soffa [13] have investigated ways to guide test-case generation, so that a single test case satisfies multiple coverage requirements. They gather coverage requirements so that each group can be covered by a single test case. Our result improves on their approach, in that spanning sets of entities provide the optimal way to group entities.[2] A different approach is to minimize the number of test cases in a test suite. Because this problem is NP-complete [13], some authors have proposed heuristics based on minimization techniques to deal with it (e.g., [14], [24]). However, these techniques are applied to a redundant set of test cases only after the test suite has been generated. Thus, such approaches do not actually reduce the effort of generating the test cases.

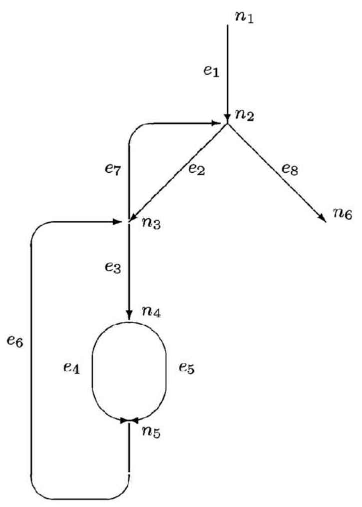
A word of caution is appropriate. By targeting test-case selection and avoiding redundant test cases, spanning sets can make coverage testing more efficient, but not necessarily more effective [22]. On the contrary, the testers should be aware that every single test case that they discard could have been the one that found the bug. Indeed, it is not our recommendation to use spanning sets to reduce the number of test cases at any rate; more pragmatically, we say that in those cases in which test resources are scarce and only few more test cases can be executed, then spanning sets can help in selecting those test cases that maximize coverage.

In some empirical studies [24], [22], the effect on fault detection of reducing the size of a test set, while holding coverage constant, was analyzed. The results in [24] showed that minimizing the test set produces little or no reduction in fault-detection effectiveness. This case study would

2. We do not actually speak explicitly in terms of groups of entities, but such groups can easily be derived by using the ordering imposed by the *subsumption* relation introduced in this paper.

```
void sort(a, n) /* SORT Program */
1  int a[];
2  int n;
3 {
4    int sortupto;
5    int maxpos;
6    int mymax;
7    int index;

8    sortupto = 1;
9    maxpos = 1;
10   while (sortupto < n) {
11     mymax = a[sortupto];
12     index = sortupto + 1;
13     while (index <= n) {
14       if (a[index] > mymax) {
15         mymax = a[index];
16         maxpos = index;
17       }
18       index = index + 1;
19     }
20     index = a[sortupto];
21     a[sortupto] = mymax;
22     a[maxpos] = index;
23     sortupto = sortupto + 1;
24   }
25 }
```



Code Lines	1-9	10	11-12	13	14	15-17	18-19	20-24	25
Ddgraph Element	e_1	n_2	e_2	n_3	n_4	e_4	e_6	e_7	e_8

Fig. 1. Program SORT and ddgraph G_{SORT}.

indicate that trying to minimize the number of test cases generated to achieve structural coverage, as we do with spanning sets, can help to reduce the cost of testing without impairing testing efficacy. The results in [22] for a different experiment, however, showed instead that test-suite minimization might produce significant reductions in the fault-detection effectiveness. Motivated by these results, we also investigated empirically the effect of using spanning sets on fault detection. We observed that, even though fault-detection effectiveness may decline slightly for high coverage values, the sizes of spanning set-based test suites were much reduced. So, if we consider the trade off between the fault-detection effectiveness and the test-suite size, then spanning-set suites seem to perform better.

In the next section, we give some background information. In Section 3, we present the coverage criteria considered in this paper, by identifying for each criterion the entities to be covered. In particular, we explicitly define the meaning of coverage for the possible types of entities. In Section 4, we define spanning sets of entities and briefly discuss several interesting applications. In Section 5, we outline a generalized method to derive a spanning set of entities. This method uses the subsumption relation between entities, which varies according to the type of entity considered. In Section 6, we provide implementations for the subsumption relation. In Section 7, we present the results of our empirical investigation of the fault-detection effectiveness of test suites constructed using spanning sets. Finally, in Section 8, we draw our conclusions and hint at future work.

2 Definitions

2.1 The Ddgraph Model

Typically, in code-based testing strategies a program's structure is analyzed on the program *flowgraph* (i.e., an annotated directed graph, or digraph, that represents graphically the information needed to select the test cases). What changes from one author's flowgraph to another's is the mapping between program entities (statements and predicates) and flowgraph elements (arcs and nodes). We use *ddgraphs* (decision-to-decision graphs).

Definition 1. *A* ddgraph *is a digraph $G = (N, A)$ with two distinguished arcs e_1 and e_k (the unique entry arc and the unique exit arc, respectively), such that any arc $e \in A$ is reached by e_1 and reaches e_k, and such that for each node $n \in N$, except $T(e_1)$ and $H(e_k)$, $(indegree(n) + outdegree(n)) > 2$, while $indegree(T(e_1)) = 0$ and $outdegree(T(e_1)) = 1$, $indegree(H(e_k)) = 1$ and $outdegree(H(e_k)) = 0$.*

Fig. 1 gives an example program SORT (adapted from [23]) and the corresponding ddgraph G_{SORT}.

We adopted the ddgraph model because it is more compact for coverage analysis than traditional flowgraphs that associate a node with each program statement or block [2]. In fact, ddgraphs do not contain any nodes that have just one arc entering and one arc leaving them: Nodes represent either *decisions* (i.e., forking of the control flow) or *junctions* (merging of the control flow); program blocks are mapped directly to arcs. In this way, the size of graphs is reduced and the control flow is immediately captured.

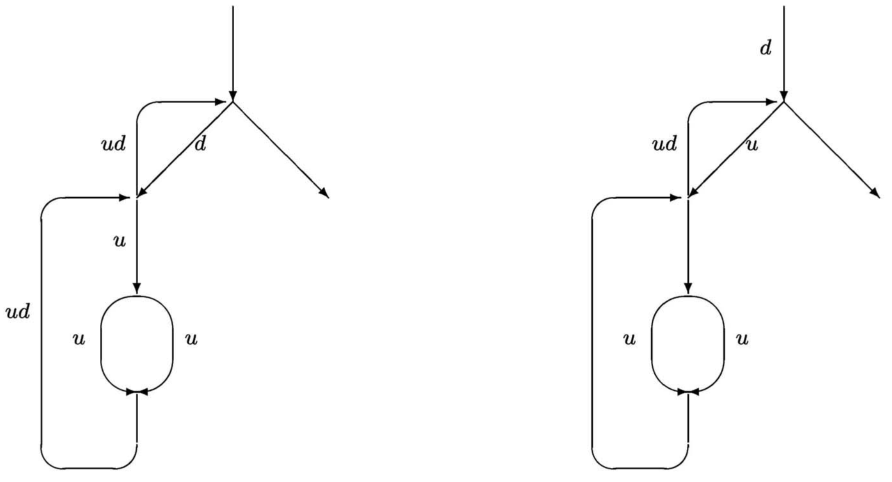


Fig. 2. Def-use ddgraphs for program SORT and variables $index$ and a, respectively.

An arc e in a ddgraph G is an ordered pair of *adjacent* nodes, called *TAIL* and *HEAD* of e, respectively (i.e., $e = (TAIL(e), HEAD(e))$). A *path* p of length q in a ddgraph G is a sequence $p = e_1, e_2, \ldots, e_q$, where $TAIL(e_{i+1}) = HEAD(e_i)$ for $i = 1, \ldots, q-1$. A path p is *simple* if all its nodes, except possibly the first and last, are distinct. A *complete path* in a ddgraph G is a path from the entry node to the exit node of G. Given a path $p = e_1, e_2, \ldots, e_q$, a path $p' = e_i, \ldots, e_j$ from e_i to e_j, with $1 \le i \le j \le q$, is called a *subpath* of p.

2.2 Def-Use Ddgraphs

Data flow testing considers the possible interactions between definitions and uses of variables. To analyze these interactions, programs are represented as annotated flowgraphs. In particular, we use annotated, or *def-use*, ddgraphs.

Given a ddgraph corresponding to a program, for every variable in the program, a *def-use ddgraph* is derived, in which each arc is annotated with a sequence (which may be empty) of the symbols d or u, to represent that the variable of interest is defined or referred in the program block represented by such arc. Note that, for a predicate use (which in our ddgraph model occurs at a decision node), we label with a use symbol u each arc leaving the node at which the predicate occurs. Fig. 2 shows two def-use ddgraphs corresponding to program SORT and variable $index$ and to program SORT and variable a.

Given a def-use ddgraph G for variable X, a *def-clear path* with respect to X is a path $p = e, e_1, e_2, \ldots, e_q, e'$ on G, with $q \ge 0$, such that X may be defined in e, and is not redefined or killed in any of the arcs $e_1, e_2, \ldots, e_q$. For example, $p_1 = e_2, e_3, e_5, e_6$ is a def-clear path for $index$ in G_{SORT} because $index$ is defined in e_2 and is not redefined or killed on e_3 or e_5. As another example, $p_2 = e_2, e_3, e_4, e_6, e_7$ is not a def-clear path for $index$ in G_{SORT} since $index$ is redefined in e_6.

In data-flow testing, we consider *global* definitions and uses (i.e., the interactions of variable assignments and usages *between* arcs). Hence, we define a definition-use association or *dua* as follows:

Definition 2. *Let d and u be two arcs in G and X be a variable. We say that the triple $[d, u, X]$ is a* definition-use association, *or a* dua, *if X has a global definition in d, a global use in u, and there is a def-clear path w.r.t. X from d to u.*

For example, $T_1 = [e_2, e_7, index]$ and $T_2 = [e_6, e_4, index]$ are duas in G_{SORT}. As another example, $T_3 = [e_7, e_4, index]$ is not a dua since $index$ is defined in e_7, $index$ is used in e_4, but there is no def-clear path w.r.t. $index$ from e_7 to e_4.

In the following, we denote the set of all the duas in a given ddgraph G as $D(G)$. The duas involving a same variable defined in a same arc of the ddgraph can be grouped into *classes*. Thus, we group into the class S_d^X all the duas in $G = (N, A)$, such that variable X is defined in arc d (i.e., $S_d^X = \{T \in D(G) : \exists u \in A, T = [d, u, X]\}$).

For example, the classes of duas for SORT and variable a are:

$$S_{e_1}^a = \{[e_1, e_2, a], [e_1, e_7, a], [e_1, e_4, a], [e_1, e_5, a]\},$$
$$S_{e_7}^a = \{[e_7, e_2, a], [e_7, e_7, a], [e_7, e_4, a], [e_7, e_5, a]\}.$$

3 A Family of Coverage Testing Criteria

Coverage criteria require that a set of *entities* of the program flowgraph is covered when the test cases are executed. For each test criterion c, we denote the corresponding set of entities of G by $E_c(G)$. In Table 1, we (re)define a family of well-known control flow and data flow test coverage criteria [5], [21], [12] by identifying, for each criterion, the set $E_c(G)$. In the table, let $G = (N, A)$ be a ddgraph and $\wp$ the set of all complete paths in G.

Below, we present some examples.

- The set of entities for the ddgraph $G_{SORT} = (N_{SORT}, A_{SORT})$ and all-branches criterion is

$$E_{\textit{all-branches}}(G_{SORT}) = A_{SORT} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}.$$

- The set of entities for the ddgraph G_{SORT} and all-statements criterion is

TABLE 1
Coverage Criterion and Entities to be Covered to Satisfy that Criterion

Coverage Criterion c	**Set $E_c(G)$ of entities for ddgraph G and coverage criterion c**
all-paths	$\{p \in \wp$: p is a complete path in $G\}$
all-k-paths	$\{p \in \wp$: p is a complete path in G and no loop in p is iterated more than k times$\}$
all-branches	A
all-statements	$\{e \in A$: e is associated with at least one instruction$\}$
all-du-paths	$\{p \in \wp$: $\exists T = [d, u, X] \in D(G)$ such that p is a simple, def-clear path wrt X from d to $u\}$
all-uses	$D(G)$
all-defs	$\{S_d^x$: $\exists T = [d', u', X'] \in D(G)$ such that $d = d'$ and $X = X'\}$

$$E_{all\text{-}statements}(G_{SORT}) = \{e_1, e_2, e_4, e_6, e_7, e_8\}.$$

- The set of entities for G_{SORT} and all-uses criterion is

$$E_{all\text{-}uses}(G_{SORT}) = D(G_{SORT}) =$$
$$\{[e_1, e_8, n], [e_1, e_2, n], [e_1, e_7, n], [e_1, e_3, n],$$
$$[e_1, e_2, a], [e_1, e_7, a], [e_7, e_2, a], [e_7, e_7, a],$$
$$[e_7, e_4, a], [e_7, e_5, a], [e_1, e_4, a], [e_1, e_5, a],$$
$$[e_1, e_2, sortupto], [e_1, e_7, sortupto], [e_7, e_8, sortupto],$$
$$[e_1, e_8, sortupto], [e_7, e_2, sortupto], [e_7, e_7, sortupto],$$
$$[e_1, e_7, maxpos], [e_4, e_7, maxpos],$$
$$[e_2, e_7, mymax], [e_2, e_4, mymax], [e_4, e_7, mymax],$$
$$[e_4, e_4, mymax], [e_2, e_5, mymax], [e_4, e_5, mymax],$$
$$[e_2, e_3, index], [e_6, e_7, index], [e_2, e_7, index],$$
$$[e_2, e_4, index], [e_6, e_4, index], [e_6, e_3, index],$$
$$[e_2, e_5, index], [e_6, e_5, index], [e_2, e_6, index],$$
$$[e_6, e_6, index]\}.$$

- The set of entities for G_{SORT} and all-defs criterion is

$$E_{all\text{-}defs}(G_{SORT}) = \{S^n_{e_1}, S^a_{e_1}, S^a_{e_7}, S^{sortupto}_{e_1}, S^{sortupto}_{e_7},$$
$$S^{maxpos}_{e_1}, S^{maxpos}_{e_4}, S^{mymax}_{e_2}, S^{mymax}_{e_4}, S^{index}_{e_2}, S^{index}_{e_6}\}.$$

In general, for the family of coverage criteria in Table 1, four different types of entities can be distinguished: *arcs*, which are the entities for all-branches and all-statements criteria; *duas*, which are the entities for all-uses; *classes of duas*, which are the entities for all-defs; and *paths*, which are the entities for all-paths, all-k-paths and all-du-paths criteria.

We observe that "covering an entity" differs depending on the type of entity considered; hence, we now define explicitly what coverage means for each of the four types of entities listed above.

Definition 3. *A complete path p covers*

- *an arc if p contains that arc;*
- *a dua $T = [d, u, X]$ if p has a def-clear subpath w.r.t. X from d to u;*
- *a class of duas S if $\exists T \in S$ such that p covers T;*[3]
- *a path p' if p' is subpath of p.*

A set of complete paths $\wp$ covers an arc (or a dua, a class of duas, or a path) if some of the paths in $\wp$ do.

For example, considering path $p = e_1, e_2, e_7, e_8$, in ddgraph G_{SORT}:

- p covers arc e_1, but does not cover arc e_3;
- p covers dua $[e_2, e_7, index]$ and also dua $[e_1, e_2, a]$, but does not cover dua $[e_2, e_3, index]$ nor dua $[e_7, e_2, a]$.

4 Spanning Sets

Given a ddgraph G and a set $E_c(G)$ of entities to be covered, it is generally possible to derive a subset of $E_c(G)$ with the property that a set of complete paths covering all entities in it will cover every c-entity in $E_c(G)$. For instance, for the ddgraph G_{SORT}, any set of complete paths covering the set of arcs:

$$\{e_3, e_4, e_5, e_6\} \subseteq E_{all\text{-}branches}(G_{SORT})$$

will cover every arc in $E_{all\text{-}branches}(G_{SORT})$. This happens because any complete path that exercises these arcs must exercise arcs e_1, e_2, e_7, and e_8 as well.

In other terms, for a selected criterion c, the coverage of some c-entities automatically guarantees the coverage of other c-entities. This property implies a natural ordering between c-entities, according to how easily they can be

3. Or, equivalently, a complete path p covers a *class of duas S* if there exists a dua $T = [d, u, X]$ in S such that p has a def-clear subpath w.r.t. X from d to u.

covered. Let us call this ordering relationship between entities a *subsumption*. The intuitive underlying idea is that if c-entity E_1 subsumes c-entity E_2, then we can "forget" about E_2 provided that we "care" about E_1.

Definition 4. *Let G be a ddgraph, c a coverage criterion, and E_1 and E_2 two entities in $E_c(G)$. Then, E_1* subsumes *E_2 if every complete path that covers E_1 covers E_2 as well.*

To make coverage testing more efficient, we are interested in identifying the maximal objects in this ordering (i.e., those entities that are not subsumed by other entities). These entities are said to be "unconstrained" (i.e., "not guaranteed"): The coverage of an unconstrained entity is not guaranteed by the coverage of any other entity. More precisely, an entity E is said to be an *unconstrained entity* if there exists no other entity E' that subsumes E without being itself subsumed by E. A smallest subset of c-entities with such a property is called a *spanning set*.

Definition 5. *Let G be a ddgraph and c a coverage criterion. A subset U of $E_c(G)$ is said to be a* spanning set of entities *for G and c if*

1. *A set of paths $\wp$ that covers every entity in U covers all the entities in $E_c(G)$.*
2. *For any set $U' \subseteq E_c(G)$, such that any set of paths that covers every entity in U' covers all the entities in $E_c(G)$, $|U| \leq |U'|$.*

Note that a spanning set of entities for a ddgraph and a coverage criterion is not necessarily unique.

How can spanning sets be useful in coverage testing? We discuss several potential applications.

Dynamic evaluation of test thoroughness. Coverage criteria are best used as hints to how and where test data are inadequate in exercising a program. In this sense, the ratio between the entities covered over the total number of entities in the program provides a measure of the thoroughness of the executed test cases. While testing is ongoing, another significant measure is provided by the ratio between the covered entities in a spanning set over all the entities in it. In fact, by measuring coverage over the spanning set, the tester can get a more precise estimate of how much testing is still necessary. For example, if there is only one uncovered entity in the spanning set, one test case that covers it will be sufficient to obtain full coverage, irrespective of the measure of coverage over the entire set of entities.

Bound on the number of test cases. Spanning sets of entities are also useful for estimating the cost of coverage testing. More precisely, the cardinality of a spanning set of entities can be used to estimate the number of test cases needed to satisfy a selected coverage criterion. In fact, the cardinality of a spanning set of entities coincides with the number of test cases needed, in the case that a different test path is taken to cover each entity in the spanning set. In this sense, this number can be regarded as a "safe" bound on the number of test cases needed to satisfy a selected strategy. This is not to say that a tester could not, or should not, find larger test suites (and clearly the more test cases the better).

Estimation of the number of test cases. One execution path may generally cover more than one unconstrained entity. Thus, the actual number of test cases depends on how the unconstrained entities are combined into complete paths. For example, we could find the theoretical minimum number of test cases by combining into complete paths as many unconstrained entities as possible. But, this minimum bound would not be useful in practice. Indeed, the more complex (i.e., the longer) a path, the more likely it is that the path is infeasible [5]. It is a commonly accepted fact that the *real* problem of any path-oriented test strategy is to derive *executable* test paths. Thus, in [9], we present a method for combining unconstrained branches to form paths that are as short as possible and introduce a *meaningful* bound to the number of test cases needed for the all-branches criterion. Following a similar approach, an entire family of *meaningful* lower bounds for estimating the cost of testing according to a family of coverage criteria could be settled.

Targeting test case selection. In practice, the coverage requirement is not usually satisfied on the first attempt with an initial test suite, and the tester needs to select more test cases. The information on achieved coverage is used by the testers for highlighting those programs parts that have been neglected by the executed test cases. In such a situation, unconstrained entities can be useful for guiding test-case selection, while keeping the cost of testing low. In fact, the generation of the additional test cases can be targeted to covering a spanning set of entities. Since a spanning set may be smaller than the whole set of entities in a program, the effort of test-case generation for incrementing coverage can be decreased considerably.

Avoiding redundant test cases. Focusing test-case generation on the coverage of a spanning set of entities helps to avoid selection of redundant paths. In fact, in the common situation where more test paths must be selected to increase coverage, the most useful paths are those that cover at least one as yet uncovered unconstrained entity. If a test path p is chosen that only covers already selected unconstrained entities, p will eventually be a redundant path.

Automating test path generation. Spanning sets of entities can also be used to help automate the generation of test paths. In [7], we present an algorithm that constructs a set of paths that covers a spanning set of arcs of a given ddgraph for the all-branches criterion. The algorithm has been generalized in [8] to build a set of paths that covers a spanning set of entities for a generic coverage criterion. As with any static path-generation method, the set of paths found by the algorithm might include infeasible paths. However, the algorithm uses a heuristic technique that should reduce the impact of this problem.

5 Finding a Spanning Set of Entities

We show how to find a spanning set of entities for a given ddgraph G and a given coverage criterion c. We use the subsumption relationship among entities, defined in Section 4. Subsumption is a preorder.[4] In fact, it is obviously reflexive and transitive. Given a ddgraph G and a coverage criterion c, we can then construct a digraph that represents the subsumption relationship. The nodes in the digraph are

4. A relationship R over a set S is said to be a *preorder* if it is reflexive (i.e., (s, s) is in R for every s in S) and transitive (i.e., if (s_1, s_2) and (s_2, s_3) are in R, then (s_1, s_3) is also in R).

the entities for G and c. There is an arc from node E_2 to node E_1 in the digraph if and only if E_1 subsumes E_2. The digraph thus obtained is called the *c-subsumption digraph* of G and is denoted by $S_c(G)$.

Clearly, if a c-subsumption digraph includes a strongly connected component[5] M, whenever an entity in M is covered by a test path, then all the entities in M are covered by this same test path (i.e., each entity in M subsumes every other entity in M). Hence, any entity in a strongly connected component M of $S_c(G)$ may be chosen to represent the whole set of entities in M. We shall refer to the entity chosen to represent a strongly connected component M as the *representative* of that component, written as $rep(M)$.

By reducing the strongly connected components of a subsumption digraph, we can obtain a directed acyclic graph. We merge all the nodes in each strongly connected component M to a single node n_M. The digraph obtained is called a *reduced* c-subsumption *digraph* and is denoted by $R_c(G)$.

We now consider the leaves of the reduced c-subsumption digraph (i.e., the nodes with no exit arc) and select one representative of each. Let U be the set of c-entities formed in this way. It can be proven that U is a spanning set of (unconstrained) entities (i.e., the following theorem holds).

Theorem 1. *Let G be a ddgraph, c be a coverage criterion, and $R_c(G)$ be the reduced c-subsumption digraph of G. Let U be a set of c-entities of $E_c(G)$ representing the leaves in $R_c(G)$ (i.e., each entity in U represents a leaf in $R_c(G)$ and all the leaves in $R_c(G)$ have a representative entity in U). Then, U is a spanning set of entities for G and c.*

The proof is given in [18].

For example, we present the unique spanning set of entities for the ddgraph $G_{SORT} = (N_{SORT}, A_{SORT})$ and all-branches criterion, $U_{\textit{all-branches}}(G_{SORT})$. It can be obtained by selecting the representatives of the leaves of the reduced *all-branches*-subsumption digraph for G_{SORT} in Fig. 3a (i.e., $U_{\textit{all-branches}}(G_{SORT}) = \{e_4, e_5\}$).

The unique spanning set of entities for the ddgraph G_{SORT} and all-statements criterion can be obtained by selecting the representatives of the leaves of the reduced *all-statements*-subsumption digraph for G_{SORT} in Fig. 3b (i.e., $U_{\textit{all-statements}}(G_{SORT}) = \{e_4\}$).

A spanning set of entities for G_{SORT} and the all-uses criterion can be obtained by selecting the representatives of the leaves of the reduced *all-uses*-subsumption digraph for G_{SORT} in Fig. 4. In this case, two spanning sets of entities exist since either dua can be chosen as the representative of the leaf containing duas $[e_2, e_5, index]$ and $[e_2, e_5, mymax]$:

$$\begin{aligned} U_{\textit{all-uses}}(G_{SORT}) = \{&[e_1, e_8, sortupto], [e_2, e_5, index],\\ &[e_2, e_4, index], [e_2, e_7, index], [e_1, e_5, a],\\ &[e_7, e_5, a], [e_1, e_4, a], [e_7, e_4, a],\\ &[e_4, e_5, mymax], [e_4, e_4, mymax]\},\end{aligned}$$

5. A strongly connected component in a digraph is a set of nodes such that the two nodes in any pair reach each other.

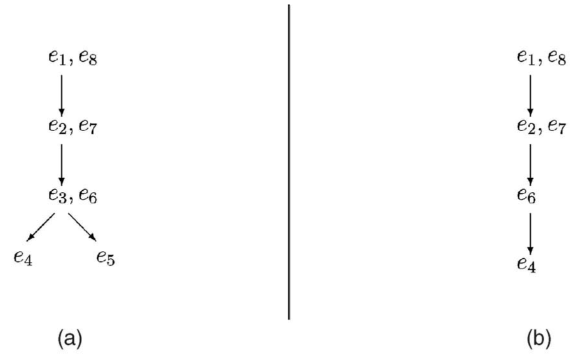


Fig. 3. Reduced subsumption digraph for G_{SORT} and all-branches and all-statements criteria, respectively.

$$\begin{aligned} U'_{\textit{all-uses}}(G_{SORT}) = \{&[e_1, e_8, sortupto], [e_2, e_5, mymax],\\ &[e_2, e_4, index], [e_2, e_7, index], [e_1, e_5, a],\\ &[e_7, e_5, a], [e_1, e_4, a], [e_7, e_4, a],\\ &[e_4, e_5, mymax], [e_4, e_4, mymax]\}.\end{aligned}$$

Note that to simplify Figs. 3a, 3b, and 4, we have drawn only the significant arcs.

The procedure FIND-A-SPANNING-SET-OF-ENTITIES $(G, E_c(G))$ below summarizes the steps for finding a spanning set of entities $U_c(G)$ for a ddgraph G and a coverage criterion c:

```
Procedure FIND-A-SPANNING-SET-OF-ENTITIES
($G$: ddgraph; $E_c(G)$: set of entities): set of
entities;
1. for each $E_1, E_2 \in E_c(G)$, $E_1 \neq E_2$,
   do SUBSUMPTION($E_1, E_2, G$);
2. construct $S_c(G) = (V_{S_c(G)}, E_{S_c(G)})$;
3. construct $R_c(G) = (V_{R_c(G)}, E_{R_c(G)})$;
4. U = {rep(M): M is a leaf of $R_c(G)$};
5. return(U).
```

The complexity analysis of FIND-A-SPANNING-SET-OF-ENTITIES is presented in [18]. It is polynomial on the number of entities (but note that it obviously becomes exponential on the number of arcs for the all-paths criterion).

6 The Subsumption Relation

In the procedure FIND-A-SPANNING-SET-OF-ENTITIES, Steps 2, 3, and 4 do not depend on the type of entities manipulated. On the other hand, evaluating whether entity E_1 subsumes entity E_2 (Step 1) depends on the notion of "coverage" associated with the particular type of entities considered (see Definition 3). Thus, we need to implement a specific SUBSUMPTION(E_1, E_2, G) procedure for each possible type of entity (arc, dua, class of duas, and path). We underline that the implementation of this procedure does not depend on the particular coverage criterion c, but only on the type of entity considered. However, the input to the procedure is the set $E_c(G)$, which is determined by c. For example, "Arc" is the type of entity associated with the all-branches and all-statements criteria; hence, we can use the same SUBSUMPTION procedure for both criteria.

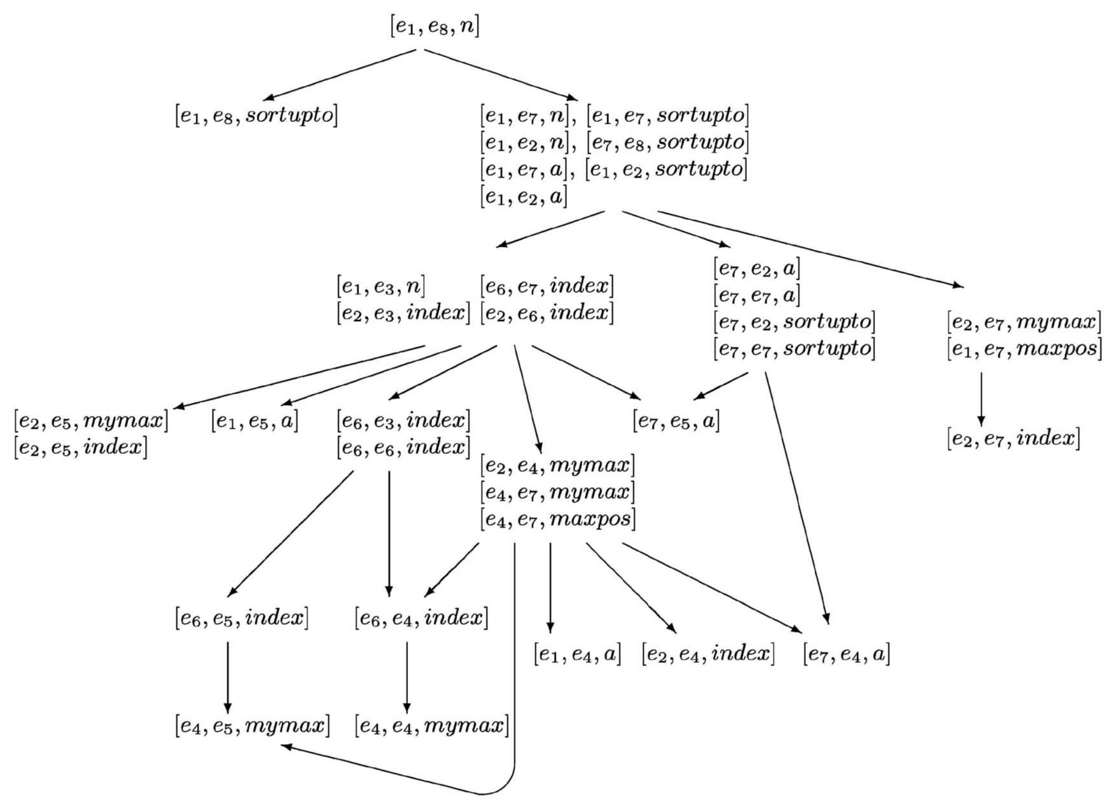


Fig. 4. Reduced subsumption digraph for G_{SORT} and all-uses criterion.

However, depending on the criterion considered, we give as an input to the procedure either $E_{all-branches}(G)$ or $E_{all\text{-}statements}(G)$.

In the following sections, we will discuss how the SUBSUMPTION procedure can be implemented when the entities considered are, respectively, arcs or duas. Subsumption between classes of duas and between paths are omitted for brevity, the interested reader is referred to [18]. We first need to provide some more definitions and concepts.

6.1 More Definitions

We introduce the notion of a *sub-ddgraph* for a given ddgraph G and two given arcs in G. We denote the sub-ddgraph of a ddgraph G from arc e_a to arc e_b as the $\text{sub-ddgraph}(G, e_a, e_b)$. Intuitively, a node (or arc) in G is in $\text{sub-ddgraph}(G, e_a, e_b)$, if there exists a path from $HEAD(e_a)$ to $TAIL(e_b)$ in G including such a node (or arc), but not including e_a or e_b. A sub-ddgraph is a ddgraph.

The sub-ddgraph G^* of a ddgraph G from arc e_a to arc e_b can be derived by visiting G. A procedure SUB-DDGRAPH which, given a ddgraph $G = (N, A)$ and two arcs e_a and $e_b \in A$, returns $G^* = \text{SUB-DDGRAPH}(G, e_a, e_b)$ in $O(|A|)$ time is given in [7]. First, we find the arcs reachable from the HEAD of arc e_a not using arc e_a. Then, from the set of arcs found in the first step, we select those that reach the TAIL of arc e_b not using arc e_b. In this process, we might obtain an intermediate digraph that is not a ddgraph because it might contain nodes with just one arc entering it and just one arc leaving it, that by Definition 1 cannot exist in a ddgraph. Therefore, the SUB-DDGRAPH procedure uses the REDUCE procedure, which transforms a digraph G' into a ddgraph G^*, by eliminating all such nodes. REDUCE eliminates each node n in the intermediate digraph G' with $indegree(n) = 1$ and $outdegree(n) = 1$ (which cannot belong to G^*) by substituting the arc e_i entering n and the arc e_j leaving n with the arc e_{i-j} in G^*. Since each arc can be reduced at most once, REDUCE can be implemented in $O(|A|)$ time.

A well-known relation from graph theory is *dominance* [15]. Dominance imposes a partial ordering[6] on the nodes or arcs of a flowgraph. In particular, we are interested in applying the dominance relation to the arcs of a ddgraph.

Given a ddgraph G and its entry arc e_1, an arc e' *dominates* an arc e if every path p on G from e_1 to e contains e'.

Several algorithms have been given in the literature to find the dominator nodes in a digraph, see, for example, [17]. Such algorithms can be easily adapted to find the dominator arcs in a ddgraph.

The *immediate dominator* e' of an arc e is a dominator of e with the property that any other dominator of e also dominates e'.

The dominance relation between the arcs of a ddgraph G can be represented by a rooted tree whose nodes represent the ddgraph arcs. This is called the *dominance tree* $DT(G)$. The root of this tree is the entry arc e_1. There exists an arc (e, e') between two nodes e and e' in the dominance tree, if e is the *immediate dominator* of e'. Note that each arc (different from e_1) has only one immediate dominator.

6. A partial ordering is a reflexive, asymmetric, and transitive relation between elements.

The *characteristic path* p_c of a ddgraph G is defined as the largest sequence of (possibly not adjacent) arcs such that, for any complete path $p = e_1, e_2, \ldots, e_q$ in G, $p_c = e_{i_1}, \ldots, e_{i_r}$, such that $\{i_1, \ldots, i_r\} \subseteq \{1, \ldots, q\}$ and for $j = 1, \ldots, r-1 : i_j < i_{j+1}$. In other words, the arcs in the characteristic path of G are included within any complete path of G.

The characteristic path p_c of G can be obtained on the dominance tree of a ddgraph G very easily. This path coincides with the path on the tree that goes from the root, representing the entry arc e_1, to the leaf representing the exit arc, e_k. We use the characteristic path to implement the subsumption relation between duas.

Next, we introduce the "symmetric" relation of dominance, called *postdominance*. This relation appears in the literature with different names (e.g., inverse-dominance [11] or implication [7]).

Given a ddgraph G and its exit arc e_k, an arc e' *postdominates* an arc e if every path p on G from e to e_k contains e'.

The postdominance relation in a ddgraph G with entry arc e_1 and exit arc e_k can be found as the dominance relation in the ddgraph G' having entry arc e'_1 and exit arc e'_k, where: each arc e' in G' is obtained by reverting a corresponding arc e in G (i.e., $HEAD(e') = TAIL(e)$ and $TAIL(e') = HEAD(e)$), e'_1 corresponds to the reverse arc of e_k and e'_k corresponds to the reverse arc of e_1.

6.2 Implementation of `SUBSUMPTION-BETWEEN-ARCS`

The procedure to check whether an arc e subsumes an arc e' (i.e., whether every complete path that covers e covers also e') is simple and exploits the following result (the proof is in [18]):

Theorem 2. *Let e and e' be two arcs in a ddgraph G, with entry arc e_1 and exit arc e_k. Then, e subsumes e' if and only if e' dominates or postdominates e.*

Therefore, in order to implement SUBSUMPTION-BETWEEN-ARCS(e, e', G), we can just check whether e' dominates or postdominates e.

6.3 Implementation of `SUBSUMPTION-BETWEEN-DUAS`

Let $T_1 = [d_1, u_1, X_1]$ and $T_2 = [d_2, u_2, X_2]$ be two duas in a ddgraph G. For space limitations, in this section, we summarize only the method used to check whether T_1 subsumes T_2 (i.e., whether every complete path that covers T_1 covers also T_2). The complete procedure is in [18]. In summary, the procedure consists of the following steps:

1. We first select all paths in G that cover T_1. We do this by constructing an intermediate ddgraph $G*$.
2. Then, we check whether every path that covers T_1 in G also traverses d_2 and u_2. This can be done by checking whether d_2 and u_2 are in the characteristic path (see Section 6.1) of $G*$.
3. Finally, to find out whether every path in $G*$ covers T_2 (i.e., not only it traverses d_2 and u_2, but is also def-clear in between), we check whether no arc e in the paths from d_2 and u_2 contains a definition of X_2.

7 Empirical Evaluation

Spanning sets are not a new test coverage criterion. They are proposed as a technique that can be incorporated within the customary procedures for coverage testing to make them more efficient, at low, if any, additional cost.

We empirically evaluated the efficiency of spanning sets within a real-world testing process, for the branch coverage criterion [10]. Beyond some difficulties with real-world constraints, the results were encouraging. In brief, in that experiment, we got two interesting conclusions. The first conclusion was that, if test-case selection is guided by the set of unconstrained branches, the performance of a beginner in terms of time employed to reach a fixed coverage is comparable to that of an expert tester. The second conclusion was that spanning sets provide a good bound for the number of test cases needed to complete coverage. How this bound is defined is presented in [9], while the experimental results are described in depth in [10].

However, the purpose of testing is to find bugs, not to reach (more or less efficiently) some coverage goal. Coverage analysis is a means to systematically sample all program portions and to discover potential weak points in a test suite. But, no compelling relationship between test coverage and test effectiveness in fault detection has been established. One natural question then is whether using spanning sets to make testing cheaper might produce, as a negative effect, a significant loss in the fault-detection effectiveness.

Some recent papers [24], [22] have investigated a related question: whether minimizing a test suite while keeping its coverage constant affects the test suite's fault-detection capability and if so to what extent. Such studies showed that the effectiveness reduction caused by minimization varies with the degree of coverage reached: It can be irrelevant for low coverage degrees, but may increase significantly when coverage is high. Our case is different, in that we do not minimize an existing test suite, but use spanning sets to generate a priori a test suite that is minimal (i.e., no test path in it can be eliminated without reducing the coverage below 100 percent).

In the cited papers, the authors compared the faults detected by the original test suite and the fraction of them that are still found by the minimized suite. Here, we should compare the sets of faults found by pairs of different test suites generated one using the spanning set approach and another without (i.e., using a conventional approach).

To address this issue, we have conducted a laboratory experiment in which we simulated the usage of spanning sets as a means to make coverage testing more efficient. As subjects of the experiment, we used the HR versions of the seven Siemens programs [22]. The description of the original programs and their use in empirical studies are given in [16]; the HR variants of these programs, that we use here, have been produced by the Aristotle Research Group (they can be obtained on line from the Aristotle Analysis System Download page [12]). In Table 2, we report some summary information of these programs and, for a more detailed description, we refer to the above cited sources. The advantage of using such subjects is that they come documented with not only many (thousands) test

TABLE 2
The Subject Programs

Program	Size in LOCs	No. of Faulty Versions	No. of Available Test Cases	Description
Print token	402	7	4130	lexical analyzer
Print token 2	483	10	4115	lexical analyzer
Replace	516	516	5542	pattern replacement
Schedule	299	9	2650	priority scheduler
Schedule 2	297	10	2710	priority scheduler
Tcas	138	41	1608	altitude separation
Totinfo	346	23	1052	information measure

cases from which the test suites that satisfy a test criterion can be drawn, but also with many realistic faults: Each program has several versions, each containing one fault. They can thus be used for evaluating the fault-detection capability of test techniques.

In the experiment, we simulated a situation in which, using an arbitrary criterion for test selection, an initial coverage C_0 has been obtained. This coverage is deemed not sufficient, and more test cases must be run until a target coverage $C_T > C_0$ is reached. To select the additional test cases, the simulation experiment proceeded along two independent observations: In one case, the baseline, the additional test cases were selected without considering spanning sets; in the second case, the additional test cases were selected so that each new test case covers one as yet uncovered unconstrained entity.

The method to use as a baseline for comparison was a critical decision. In practice, there exists no standard approach for coverage testing, and test derivation is done mostly manually. In such a context, it is likely (and it has also been empirically observed) that the tester's expertise and intuition can considerably influence the performance of the testing. To obtain generally valid results, we decided to draw the additional test cases at random. This decision can be justified by the following reasoning: as we want to assess in the experiment the loss in fault-detection effectiveness caused by the reduction of the number of test cases, then a random approach to test-case selection provides the worst-case base of comparison because, for a fixed coverage, it obviously provides on average larger test suites than those that would be provided by a tester.

For each of the two cases, we measured:

- How many test cases were taken cumulatively to obtain coverage C_T (i.e., the initial test cases plus the additional ones); we call such a measure the test size, or TS.
- How many faults were found in total, or NF.

We ran the experiment considering for C_T several progressive increments of 5 percent from C_0, with $C_0 = 50$ percent, up to $C_T = 100$ percent. That is, we measured TS and NF at $C_T = 55$ percent, $C_T = 60$ percent, etc. For each value of C_T, we repeated the observation several times to obtain in the two cases (with and without spanning sets) mean values for TS and NF that were statistically valid within a 95 percent confidence interval.

The simulation experiment was conducted for the two criteria of all-branches and all-uses coverage. The detection of spanning sets was done partly with proof-of-concept tools, and partly manually (for the duas). Therefore, the results for spanning sets of duas were obtained only for five of the subject programs. In Tables 3 and 4, we show the measures obtained at 65, 75, 85, 90, and 95 percent of coverage for the two experiments (all-branches and all-uses, respectively). The tables include three pairs of columns: In each pair, SS refers to the observations with spanning sets and RA to the observations without spanning sets. The first pair of columns gives the total number of faults detected NF, the second pair the cumulative number of executed test cases TS, and the third pair refers to the fault-detection density, denoted by FD. FD is obtained as the ratio[7] between the faults found and the number of test cases. FD can be regarded as a rough measure of the effectiveness of the test suites (with the caution that then we are comparing the effectiveness of a random approach vs. spanning sets). All the values in the tables are mean values significant at 95 percent of the confidence interval. We applied the F-test (well-known in statistics [25]) to analyze the significance of the showed differences; when the differences were statistically significant (i.e., according to the F-test), the relative pairs of cells are reported with light-gray background; the white cells did not show significant differences.

The results are quite consistent with previous studies [24], [22]: In fact, for coverage levels below 85 percent, no (statistically) significant difference in the number of faults

7. The ratio is calculated on the single observations and not between the mean values of NF and TS shown in the tables.

TABLE 3
Results for the All-Branches Experiment

	NF		TS		FD	
C_T=65%	SS	RA	SS	RA	SS	RA
Print token	0.67	0.67	5.93	6.63	0.13	0.12
Print token 2	1.63	1.63	2.73	2.87	0.82	0.82
Replace	1.27	1.43	3.9	4.17	0.40	0.41
Schedule	0.43	0.43	1.63	1.63	0.28	0.28
Schedule 2	0.2	0.2	1.77	1.43	0.13	0.17
Tcas	3.5	4.5	3.5	4.9	1.08	0.97
TotInfo	2.17	2.17	1.3	1.3	1.76	1.76
C_T=75%	SS	RA	SS	RA	SS	RA
Print token	0.97	1.03	8.17	11.43	0.14	0.12
Print token 2	2.17	2.23	3.73	4.37	0.81	0.78
Replace	1.87	2.2	5.8	7.07	0.34	0.32
Schedule	0.63	0.63	1.87	1.9	0.34	0.34
Schedule 2	0.33	0.33	1.77	1.8	0.22	0.21
Tcas	3.97	5.1	3.83	5.63	1.072	0.91
TotInfo	4.7	4.7	2.53	2.97	2.38	2.21
C_T=85%	SS	RA	SS	RA	SS	RA
Print token	1.7	2	10.83	19.9	0.17	0.12
Print token 2	2.63	2.77	5.3	6.7	0.66	0.62
Replace	3.1	5.63	9.53	17.93	0.33	0.32
Schedule	0.73	0.73	2.37	2.43	0.34	0.34
Schedule 2	0.93	0.97	3.3	5.4	0.31	0.26
Tcas	5.4	7.67	4.37	8.8	1.29	0.95
TotInfo	8.03	8.23	4.03	5.63	2.21	1.84
C_T=90%	SS	RA	SS	RA	SS	RA
Print token	2.13	2.53	12.5	32.67	0.18	0.09
Print token 2	3.4	3.67	6.87	10.3	0.57	0.44
Replace	4.3	8.93	12.57	36.13	0.34	0.25
Schedule	1.07	1.37	3.63	6	0.33	0.30
Schedule 2	2	2.4	4.83	20.17	0.44	0.22
Tcas	8.13	13	5.3	16.37	1.59	0.98
TotInfo	11.2	11.7	5.73	9.77	2.04	1.43
C_T=95%	SS	RA	SS	RA	SS	RA
Print token	3.3	7	9.07	21.93	0.21	0.00
Print token 2	5.7	6.07	5.7	6.07	0.70	0.35
Replace	5.63	18.8	16.97	113.77	0.33	0.18
Schedule	2.37	5.4	6.43	64.2	0.40	0.12
Schedule 2	2.6	6.07	7.47	140.17	0.36	0.05
Tcas	8.93	22.2	6.3	48.8	1.45	0.58
TotInfo	12.23	23	6.9	1052	1.80	0.02

TABLE 4
Results for the All-Uses Experiment

	NF		TS		FD	
C_T=65%	SS	RA	SS	RA	SS	RA
Replace	1.47	1.6	4.67	5.3	0.30	0.30
Schedule	0.47	0.47	1.53	1.53	0.28	0.28
Schedule 2	0.2	0.2	1.4	1.43	0.17	0.17
Tcas	4.5	4.63	4.17	4.97	1.09	0.93
TotInfo	1.5	1.57	2.47	2.3	1.91	1.82
C_T=75%	SS	RA	SS	RA	SS	RA
Replace	2.23	2.77	7.37	9.3	0.30	0.29
Schedule	0.5	0.5	1.7	1.7	0.28	0.28
Schedule 2	0.43	0.43	2.27	2.4	0.19	0.19
Tcas	7.1	6.6	5.23	7.23	1.37	0.96
TotInfo	3.23	4.6	7.07	6.87	2.54	2.05
C_T=85%	SS	RA	SS	RA	SS	RA
Replace	4.07	7.97	13.83	31.1	0.31	0.27
Schedule	0.7	0.7	2.17	2.37	0.34	0.34
Schedule 2	1.9	3.43	5.23	34.33	0.39	0.14
Tcas	9.57	11.37	6.73	12.73	1.43	0.97
TotInfo	5.23	23	10.87	1052	2.19	0.02
C_T=90%	SS	RA	SS	RA	SS	RA
Replace	5.63	18.9	18.93	111.83	0.30	0.18
Schedule	1.03	0.8	2.63	2.93	0.42	0.33
Schedule 2	1.9	9	5.43	2710	0.36	0.00
Tcas	10.07	16.33	7.37	22.8	1.39	0.81
TotInfo	5.23	23	10.87	1052	2.19	0.02
C_T=95%	SS	RA	SS	RA	SS	RA
Replace	5.63	32	18.93	5542	0.30	0.01
Schedule	1.87	4.9	4.73	58.37	0.41	0.11
Schedule 2	1.9	9	5.43	2710	0.36	0.00
Tcas	10.07	41	7.4	1608	1.39	0.02
TotInfo	5.23	23	10.87	1052	2.19	0.02

found with and without spanning sets is observed. However, as also observed in [24], [22] relatively to minimization, as coverage increases, the difference in the number of faults found grows (in favor of random selection), although only at a coverage C_T of at least 95 percent these differences become statistically significant for almost all subject programs (except Print token 2). On the other hand, the random approach takes on average a higher number of test cases to reach the same coverage as the spanning set approach, with most differences already statistically significant from $C_T = 85$ percent. Notably, if we look at the fault-detection density, we see that it does not change significantly until 90 percent in Table 3 and 85 percent in Table 4 and for higher values is significantly better for spanning sets. So, for instance, in branch coverage, we observe that at $C_T = 95$ percent the spanning-set suites for Replace find on average 5.6 faults, against the 18.8 found by the random suite, but the former consists on average of 16.9 test cases, while the latter of 113.7 test cases. In fact, FD for this example holds 33.0 percent for SS, against the 18.2 percent for RA.

These results seem quite encouraging. We started the experiment to investigate whether spanning sets produce a significant loss in fault detection. The results we obtained showed that there is no relevant loss when coverage is below 85 percent, while, for higher values, the loss increases, but, at the same time, the number of test cases employed for the random approach grows so much that the fault-detection density per test case either remains the same, or is even higher for spanning sets.

What does this mean in practice? The message is NOT that doing fewer test cases is better than doing more. All the test cases that can be afforded should be done because any single test case could be useful to discover new bugs. However, the results observed for the fault-detection density seem to imply that if only a limited number of test cases can be executed, then, by using spanning sets, we can obtain test suites that are more effective not only at improving coverage, but also at finding faults. Clearly, further experimentation is needed to generalize these conclusions.

8 Conclusions and Future Work

We have introduced the concept of a spanning set of entities for coverage testing (i.e., a minimum subset U of entities such that coverage of U guarantees coverage of the entire set). We have provided a method to derive a spanning set that is parameterized in the subsumption relation between entities. The method uses simple static analysis techniques and is general for the entire family of coverage criteria. We have suggested several applications of spanning sets. In particular, they are useful for reducing and estimating the number of test cases and for evaluating test-suite thoroughness more effectively. We have also investigated empirically the fault-detection effectiveness of test suites derived using spanning sets.

This paper is meant to introduce the novel notion of spanning sets into the testing theory literature. Many interesting extensions can be foreseen on the research side.[8] Indeed, the development of spanning set theory at the unit level of testing is merely the first step. The real payoff can be perceived with the application of spanning sets to the interprocedural level; in other words, how the notion of spanning sets can be extended to reduce and estimate system-level regression test suites.

Much work also remains to be done to validate the practical value of our results. It is our hope that testing practitioners, by incorporating our spanning set-based techniques into their test processes, will want to continue gathering more empirical evidence to confirm the usefulness of spanning sets.

References

[1] H. Agrawal, "Dominators, Super Blocks, and Program Coverage," *Proc. Principles of Programming Languages (POPL '94),* pp. 25-34, Jan. 1994.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] Aristotle Research Group, "Aristotle Analysis System Download," http://www.cc.gatech.edu/aristotle/Tools/dist.html, 2003.

[4] K. Beck, *Test Driven Development: By Example.* Addison-Wesley, 2002.

[5] B. Beizer, *Software Testing Techniques, Second Edition.* New York: Van Nostrand Reinhold, 1990.

[6] A. Bertolino, "Unconstrained Edges and Their Application to Branch Analysis and Testing of Programs," *The J. Systems and Software,* vol. 20, no. 2, pp. 125-133, Feb. 1993.

[7] A. Bertolino and M. Marré, "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs," *IEEE Trans. Software Eng.,* vol. 20, no. 12, pp. 885-899, Dec. 1994.

[8] A. Bertolino and M. Marré, "A General Path Generation Algorithm for Coverage Testing," *Proc. Int'l Quality Week,* May 1997.

[9] A. Bertolino and M. Marré, "How Many Paths Are Needed for Branch Testing?" *The J. Systems and Software,* vol. 35, no. 2, pp. 95-106, Nov. 1996.

[10] A. Bertolino, R. Mirandola, and E. Peciola, "A Case Study in Branch Testing Automation," *The J. Systems and Software,* vol. 38, no. 1, pp. 47-59, July 1997.

[11] T. Chusho, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," *IEEE Trans. Software Eng.,* vol. 13, no. 5, pp. 509-517, May 1987.

[12] P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1483-1498, Oct. 1988.

[13] R. Gupta and M.L. Soffa, "Employing Static Information in the Generation of Test Cases," *Software Testing, Verification and Reliability,* vol. 3, no. 1, pp. 29-48, 1993.

[14] M.J. Harrold, R. Gupta, and M.L. Soffa, "Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, pp. 270-285, July 1993.

[15] M.S. Hecht, *Flow Analysis of Computer Programs.* New York: North Holland, 1977.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng. (ICSE 16),* pp. 191-200, 1994.

[17] T. Lengauer and R.E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Trans. Programming Languages and Systems,* vol. 1, no. 1, pp. 121-141, 1979.

[18] M. Marré, "Program Flow Analysis for Reducing and Estimating the Cost of Test Coverage Criteria," PhD Thesis, Dept. Computer Science, FCEyN, Univ. de Buenos Aires, 1997.

[19] M. Marré and A. Bertolino, "Unconstrained Duas and Their Use in Achieving All-uses Coverage," *ACM Proc. Int'l Symp. Software Testing and Analysis (ISSTA 96),* pp. 147-157, Jan. 1996.

[20] M. Marré and A. Bertolino, "Reducing and Estimating the Cost of Test Coverage Criteria," *Proc. 18th Int'l Conf. Software Eng. (ICSE 18),* pp. 486-494, Mar. 1996.

[21] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367-375, Apr. 1985.

[22] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong, "Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proc. Int'l Conf. Software Maintenance,* pp. 34-43, Nov. 1998.

[23] W. E. Wong, "On Mutation and Data Flow, A Thesis," PhD Thesis, SERC-TR-149-P, Software Eng. Research Center, Purdue Univ., Dec. 1993.

[24] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proc. 17th Int'l Conf. Software Eng. (ICSE 17),* pp. 41-50, Apr. 1995.

[25] T.H. Wonnacott and R.J. Wonnacott, *Introductory Statistics,* fifth ed. Wiley, 1990.

8. The following comment is gratefully attributed to Dr. Beizer (private communication).

Martina Marré received the MS degree in computer science from Escuela Superior Latinoamericana de Informática (ESLAI) (1991), and the PhD degree in computer science from University of Buenos Aires (1997). From 1995 to 2002, she was an assistant professor of computer science in the Department of Computer Science at the University of Buenos Aires. She has authored several papers that have appeared in international conferences and journals. Her current research interests include data quality, software testing, and software metrics. She is active in technology transfer to industry. Since 1997, she has worked as a consultant, specializing on data quality and software testing.

Antonia Bertolino is a researcher with the Institute of Information Science and Technologies (ISTI) of the Italian National Research Council (CNR), in Pisa, Italy, where she leads the software engineering group and the Pisatel Laboratory. Her research interests are in software engineering, especially software testing and dependability. Currently, she investigates approaches for systematic integration test strategies, for architecture and UML-based test approaches, and for component-based software analysis and testing. She is an associate editor of the *Journal of Systems and Software* and of the *IEEE Transactions on Software Engineering*. She was the general chair of the ACM Symposium on Software Testing and Analysis ISSTA 2002 and of the Second International Conference on Achieving Quality in Software AquIS '93. She has served on the program committees of several conferences and symposia, including ISSTA, Joint ESEC-FSE, ICSE, SEKE, Safecomp, and Quality Week. She has (co)authored more then 60 papers in international journals and conferences.