SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2000; **10**: 149–170

State generation and automated class testing

Thomas Ball[1,‡], Daniel Hoffman[2,*,†], Frank Ruskey[2,§], Richard Webber[3,¶] and Lee White[4,||]

[1]*Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.*
[2]*Department of Computer Science, University of Victoria, P.O. Box 3055 MS7209, Victoria, B.C., V8W 3P6 Canada*
[3]*Department of Computer Science, University of Melbourne, Gratton Street, Parkville 3052 Australia*
[4]*Department of Computer Engineering and Science, Case Western Reserve University, Olin Building, 10900 Euclid Ave., Cleveland, OH 44106-7071, U.S.A.*

SUMMARY

The maturity of object-oriented methods has led to the wide availability of container classes: classes that encapsulate classical data structures and algorithms. Container classes are included in the C++ and Java standard libraries, and in many proprietary libraries. The wide availability and use of these classes makes reliability important, and testing plays a central role in achieving that reliability. The large number of cases necessary for thorough testing of container classes makes automated testing essential. This paper presents a novel approach for automated testing of container classes based on combinatorial algorithms for state generation. The approach is illustrated with black-box and white-box test drivers for a class implemented with the red–black tree data structure, used widely in industry and, in particular, in the C++ Standard Template Library. The white-box driver is based on a new algorithm for red–black tree generation. The drivers are evaluated experimentally, providing quantitative measures of their effectiveness in terms of block and path coverage. The results clearly show that the approach is affordable in terms of development cost and execution time, and effective with respect to coverage achieved. The results also provide insight into the relative advantages of black-box and white-box drivers, and into the difficult problem of infeasible paths. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: automated testing; object-oriented; black-box; white-box; C++

*Correspondence to: Daniel Hoffman, Department of Computer Science, University of Victoria, P.O. Box 3055 MS7209, Victoria, B.C., V8W 3P6 Canada
†E-mail: dhoffman@csr.uvic.ca
‡E-mail: tball@microsoft.com
§E-mail: fruskey@csr.uvic.ca
¶E-mail: rwebber@cs.mu.oz.au
||E-mail: leew@alpha.ces.cwru.edu

Received 12 May 1999
Accepted 1 June 2000

1. INTRODUCTION

With object-oriented methods, productivity and reliability can be vastly improved, primarily through reuse. Container classes offer significant potential for reuse, giving programmers access to efficient implementations of data types such as set, vector and map. Container classes are now provided in the C++ and Java standard libraries, and in many proprietary libraries as well. With such widespread availability, the cost of errors is high, making reliability important.

Because container classes are usually implemented using ‘textbook’ algorithms and data structures, they do not suffer from the flagrant requirements and usability problems so common in today’s application software. None the less, container implementations do frequently contain faults, as an earlier study [1] and an online bug database [2] clearly demonstrate. Further, because programmers expect container failures to be rare, an application failure caused by a container failure is expensive to fix; the debugging takes a long time because the last place checked is the container implementation.

1.1. The need for automation in container class testing

Significant automation is essential for thorough container class testing due to the large number of test cases required and the need to rerun the test suites repeatedly. Thorough testing will involve many internal states, many method calls, and many values for each method parameter. Because the combinations of these state and parameter values are important, thorough test suites will routinely involve tens of thousands of test cases. As a result, automated input generation and output checking are essential. In conventional testing, the expected output is often stored in a ‘gold file’, manually checked for correctness. In each test run, the actual output is captured in another file and compared automatically with the gold file. The gold file approach is typically infeasible for container testing: the gold files are simply too large to be checked manually.

Each test suite for a class will be run not once, but many times: for the initial version, after each modification, and in each distinct environment, such as the new version of an operating system or compiler. Ideally, a change in environment should not require retesting of a class. With current technology, however, class behaviour *is* often affected by environment changes. For example, there is considerable variation across C++ compilers, especially regarding templates, exceptions and name spaces, which are key features for class libraries. Classes that run correctly with one compiler may not even compile with another.

1.2. Class testing based on state generation

While each container class has its own unique testing needs, a strategy that produces effective test drivers for many container classes is shown in Figure 1. From the vast number of possible states, the tester chooses to cover a much smaller set S of ‘base states’ likely to expose faults in the class C under test. The choice of base states depends on the data structures and algorithms in the class under test. The tester must determine a way to generate each state s in S and to check whether C behaves properly in s. The latter is accomplished

```
For each state $s$ in some set $S$ of states
    Generate $s$ in the class-under-test $C$
    Check that $C$ behaves properly in state $s$
        Use get calls to query $s$
        Use set calls that perturb $s$ slightly, yielding $s'$
        Use get calls to query $s'$
```

Figure 1. A generic component testing strategy.

with 'set calls' that perturb s slightly, yielding s', and 'get calls' that query s and s' without changing them. To apply this strategy to a particular class, three tasks must be carried out.

(1) Generate the set of base states S.
(2) Determine the set calls to be issued.
(3) Develop a *test oracle*: the code to check the get call return values from s and s'.

For task (1), an algorithm must be found in the literature or custom developed. The algorithm should be efficient enough that the test execution time is dominated by set and get call invocations, i.e. so that state generation is not the bottleneck in the test process. Tasks (1) and (2) must be solved with task (3) clearly in mind. Otherwise, the oracle will be too expensive to develop. Without a practical oracle, a generation algorithm is of no value to the tester.

1.3. `IntSet`: the class-under-test

To illustrate the approach shown in Figure 1, a number of test drivers are presented for the **`IntSet`** class, whose public member functions are shown in Figure 2. The constructor, copy constructor, and destructor provide the obvious functionality. The call **`add`**(i) adds i to the set and has no effect if i is already present. The call **`remove`**(i) removes i from the set and is ignored if i is not present. Finally, **`isMember`**(i) returns true or false according to whether i is present. All of the **`IntSet`** implementations tested use a red–black tree data structure and the add and remove algorithms described by Cormen, Leiserson and Rivest [3]. While red–black trees are a textbook data structure, the published algorithms contain important differences in the approaches used for node insertion and deletion.

```
class IntSet {
public: IntSet();
        IntSet(const IntSet&);
        ~IntSet();
        void add(int);
        void remove(int);
        int isMember(int) const;
};
```

Figure 2. Class declaration for the class-under-test.

1.4. Black-box and white-box testing

Black-box and white-box testing differ in two important ways: the motivation for selecting test cases and the way the test drivers access the code-under-test.

- *Test-case selection*. In black-box testing, test case selection is based solely on the specification. In contrast, white-box test case selection is influenced by the implementation data structures and algorithms. For example, the same black-box test suite would be used both for an **`IntSet`** red–black tree implementation and for an unordered linked-list implementation. A white-box test suite, however, would differ significantly for these two implementations, e.g. with a red–black tree, insertion order is important.
- *Test-driver access*. In black-box testing, the driver accesses the class-under-test (CUT) strictly through the public member functions. White-box tests, however, may involve direct access to private data, or modification of the implementation. For example, a white-box test may add member functions solely for test purposes.

Generally, black-box tests are simpler to implement and maintain. White-box tests are more complex and expensive, but may provide more effective testing.

The approach is illustrated with several test drivers for the **`IntSet`** class. Two basic approaches to state generation (task(1)) are explored: black-box approaches that generate states by adding integers to the set by method calls, and white-box approaches that generate states by a combinatorial algorithm that generates every red–black tree with N nodes. In both approaches, the set elements are the same—$\{2,4,\ldots,2N\}$. These values were chosen for two reasons: they are easy to generate and there is room between each pair of elements to add a new element in task (2). Tasks (2) and (3) are handled identically in all the drivers; in task (2), **`add`** and **`remove`** are called for every element in $[2..2N]$, and in task (3) every element in $[2..2N]$ is checked for membership.

While exhaustive testing is typically impossible, this approach is quasi-exhaustive in the sense that all test cases of a certain type are executed, e.g. all red–black trees with N nodes. As expected, the number of test cases is often large, frequently exceeding 100 000 cases, where each get call is considered a test case. Because container classes are usually efficient, if the generation algorithm is also efficient then the test execution time is still affordable.

The testing approach provides a way to overcome these problems and to apply the results of combinatorial generation theory to practical testing problems. The practitioner need not understand the generation algorithm or its complexity analysis, only the output. The theoretician needs to know only the data structure in question.

1.5. Paper organization

Section 2 surveys the relevant research. The remaining sections present and evaluate the test drivers. For the black-box drivers, state generation is straightforward. For the white-box drivers, however, this is a complex task. Section 3 presents the only known algorithm for generating all red–black trees with a given number of nodes, and shows that the algorithm is efficient in the sense that the cost of testing will be dominated by test execution rather than state generation. While the detailed analysis in Section 3 is important, the remainder of the paper does not depend on these details. Section 4 describes a white-box test driver based on red–black tree generation, and two black-box drivers used for comparison purposes. Section 5 presents the results of running all three drivers, measuring both block and path coverage. For the white-box driver, test runs were made with all trees of a given size and with much smaller random samples.

2. RELATED WORK

Research in software test automation can be divided into three broad approaches.

(1) Given some source code, automatically generate test inputs to achieve a given level of statement, branch or path coverage.
(2) Given a formal specification, automatically generate test cases for an implementation of that specification.
(3) Given the tester's knowledge of some source code and its intended behaviour, develop algorithms that automatically generate inputs and check outputs.

The first approach relies on identifying program paths satisfying the desired coverage criteria, and calculating a predicate associated with each path. Then constraint-solving techniques are used to generate a test case for each path. Early work by Clarke [4] and more recent work by Bertolino and Marré [5], Gotlieb *et al*. [6] and many others illustrates the approach. While considerable progress has been made over the years, the analysis is hampered by the presence of pointers, function calls, and complex expressions in predicates. Even if these problems can be overcome, this approach does not address output checking at all, which is a significant drawback. For example, the JTest tool [7] is claimed to generate test inputs that achieve 100 per cent block coverage for Java classes. Output checking is limited, however, to detecting whether any undeclared exceptions have been thrown. Recent work in selective regression testing [8] has achieved significant progress. Given a regression test suite and two versions of a program, these techniques automatically detect which test cases are guaranteed to produce the same output on both versions.

When a formal specification is available, output checking can be automated as well as input generation. Early work by Bougé *et al*. [9] and Gannon *et al*. [10] and more recent work by Hughes and Stotts [11] and Chen *et al*. [12] shows how to generate test cases from an algebraic specification. Dillon and Ramakrishna [13] show how to generate an oracle from a temporal logic specification. Stocks and Carrington present a test framework for generating test cases from Z specifications [14]. While these approaches offer great potential for automation, they are hard to apply in practice because formal specifications for industrial software are usually unavailable. For example, to the best of the current authors' knowledge, there are none for the C++ or Java standard class libraries.

When testing is based on the tester's knowledge, rather than on source code or a formal specification, some opportunities for automation are lost. This approach does, however, avoid many of the problems of the previous two approaches, and has seen considerable practical application. Early work by Panzl [15] provided tools for the regression testing of Fortran subroutines. The PGMGEN system [16] generates drivers for C modules from test scripts and the Protest system [17] automates the testing of C modules using test scripts written in Prolog. The ACE tool [18] supports the testing of Eiffel and C++ classes, and was used extensively for generating drivers for communications software. The ClassBench framework [19] generates tests by automatically traversing a *testgraph*: a graph representing selected states and transitions of the class under test. The category-partition method [20] provides a formal language for specifying input domains and a generator tool to produce test cases. Commercial test data generators are widely available, such as TDGEN [21], which generates test files based on descriptions of the allowable values in each field in a record. Considerable work has been done on statistical testing by Cobb and Mills [22], Musa *et al*. [23] and others. Here, test case selection is driven by expected usage patterns and test results include reliability estimates as well as failure reports.

While the research just described is broadly related to this paper, work in two research areas is quite closely related. Section 3 presents a new algorithm for generating red–black trees. There are algorithms for generating many other kinds of data structures and discrete objects, such as binary trees [24], B-trees [25] and partitions of a finite set [26]. Using the methodology presented in this paper, these algorithms could be used to test applications in the area of expression trees, databases and data structures for disjoint sets.

Doong and Frank [27] develop black-box drivers and use algebraic specifications for output checking. Their drivers share the following characteristics with the drivers in this paper:

- container classes are tested;
- input generation and output checking are automated;
- there are large numbers of test cases;
- the drivers are parametrized; and
- experiments are performed exploring the test space and comparing parameter values and test effectiveness.

There appears to be no other work in the literature presenting drivers and experiments like these.

3. EFFICIENT RED–BLACK TREE GENERATION

This section describes an algorithm for generating red–black trees and proves that it is efficient. This is done by first developing an efficient algorithm for generating 2-3-4 trees and then adapting that algorithm to generate all red–black trees.

3.1. Red–black and 2-3-4 trees

A binary search tree is a binary tree in which every node has an associated key, and such that an inorder traversal of the tree visits the keys in non-decreasing order (see Reference [3], Chapter 13). Binary search trees are widely used in computer science, but suffer from poor worst-case search times due to the imbalance that can occur in the structure of the tree. Various schemes, such as height-balanced trees, have been proposed to alleviate this worst-case behaviour, and currently the most popular is the so-called red–black tree. Red–

black trees are extended binary search trees that, in addition to the usual requirement that keys in left subtrees are no greater than those in the corresponding right subtrees, satisfy the following four properties.

(1) Every node is coloured either red or black.
(2) Every leaf node is coloured black.
(3) If a node is red, then both of its children are black.
(4) Every path from the root to a leaf contains the same number of black nodes. This number is called the black-height of the tree.

Figure 3 shows a red–black tree whose keys are the first six prime numbers. Black internal nodes are coloured black, red internal nodes are shaded, and the leaves (think of them as being coloured black by rule (2)), are shown with squares. These four conditions are sufficient to ensure that the height of the tree is $O(\log n)$, where n is the number of nodes in the tree [3].

A 2-3-4 tree is a B-tree of order 4 ([3], p. 385). In other words, it is an ordered tree in which every leaf occurs at the same level and every non-leaf node has 2, 3 or 4 children. There is an intimate connection (actually, a simple many-to-one mapping) between red–black and 2-3-4 trees that allows one to transform an algorithm for generating 2-3-4 trees into an algorithm for generating all red–black trees. This connection will be explained in Section 3.3 after the presentation of an algorithm for generating all 2-3-4 trees.

3.1.1. Generating 2-3-4 trees

Let $\mathbf{B}(n)$ denote the set of all 2-3-4 trees with n leaves. In order to generate the set $\mathbf{B}(n)$ recursively, it is convenient to generalize the set by adding another parameter. Let $\mathbf{B}(s, d)$ be the set of subtrees of 2-3-4 trees such that all leaves are at one of two adjacent levels, and where all leaves at the last level (of which there are s) occur before any at the

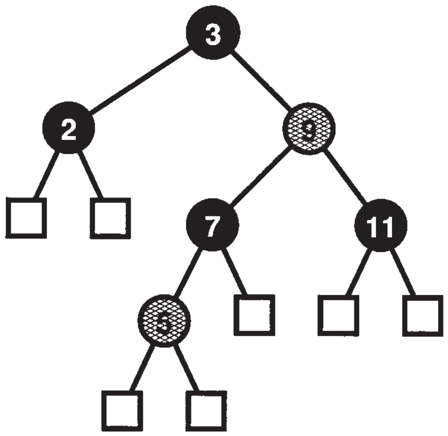


Figure 3. A typical red–black tree.

penultimate level (of which there are d), in a preorder traversal. All non-leaf nodes must have 2, 3 or 4 children. The last internal node in a level-order traversal will be called the *critical* node. In Figure 4, the critical node is shaded.

Thus $\mathbf{B}(n,0)$ is the same as the set $\mathbf{B}(n)$. The numbers $B(s,d) = |\mathbf{B}(s,d)|$ satisfy recurrence relation (1) below. This relation, for B-trees, first appeared in a work by Kelsen [25].

$$B(s,d) = \begin{cases} 1 & \text{if } s = 1 \\ B(d+1,0) & \text{if } s = 2,3 \\ B(d+1,0) + B(2,d+1) & \text{if } s = 4 \\ B(2,d+1) + B(3,d+1) & \text{if } s = 5 \\ B(s-2,d+1) + B(s-3,d+1) & \\ \qquad + B(s-4,d+1) & \text{if } s \geq 6 \end{cases} \tag{1}$$

Each case of this recurrence relation is fairly easy to understand by considering the underlying trees. Clearly, if $s = 1$, then there is only one tree, namely the tree with one node. For $s = 4$, note that $B(4, d) = B(d+1,0) + B(2, d+1)$ because the critical node must either have four children (and removing those children yields a tree in $\mathbf{B}(d+1,0)$) or have two children (and removing those children yields a tree in $\mathbf{B}(2,d+1)$). Similarly, if $s = 2$ or $s = 3$, then the children of the critical node must be all nodes at the lowest level; in these cases $B(s,d) = B(d+1,0)$. In the general case of $s \geq 6$, simply classify the trees according to the number $i = 2, 3, 4$ of children of the critical node; there are $B(s-i, d+1)$ trees in which the critical node has i children. For $s = 5$ the critical node cannot have 4 children. This discussion proves the recurrence relation (1).

A new encoding scheme for 2-3-4 trees is now introduced. Each 2-3-4 tree, and more generally, the trees of $\mathbf{B}(s,d)$, can be encoded by recording, in a reverse level-order traversal, the number of children of each node, ignoring the leaves. A *reverse level-order traversal* is a traversal that is done from greatest to smallest depth and right-to-left at each depth. This sequence of numbers, one for each internal node, is referred to as the *tree sequence* of the tree. For example, the two tree sequences for $n = 5$ are 232 and 322; for $n = 6$ the sequences are 242, 332, 422 and 2223. Figure 4 shows a typical tree in $\mathbf{B}(6,3)$ and its corresponding tree sequence. Given a tree sequence, the originating tree can easily be constructed by starting at the root of the tree, processing the sequence from right-to-left.

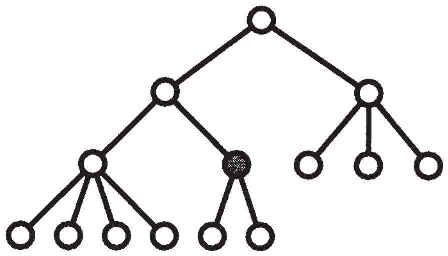


Figure 4. A typical tree in $\mathbf{B}(6,3)$; its tree sequence is 24322.

The reasoning used to justify the recurrence relation for $B(s,d)$ can be used to derive a simple recursive program for generating tree sequences. The C program of Figure 5 will output all 2-3-4 tree sequences; note how it closely follows the structure of equation (1). It is called **`ttf`** for *t*wo, *t*hree, *f*our. The initial call is **`ttf`**(n, 0, 0); the array **`a`** need not be initialized. The procedure **`Printit(p)`** processes the array, which now holds the tree sequence in positions **`a[0 . .p–1]`**.

3.2. Analysing the efficiency of the algorithm

The amount of computation used to generate $\mathbf{B}(n,0)$ is proportional to the number of recursive calls to **`ttf`**, since only a constant amount of computation is done for any given recursive call and every recursive call eventually leads to the generation of at least one tree sequence. Of interest is the ratio of the number of nodes in the computation tree (i.e. the number of recursive calls) to the number of leaves (i.e. the number of 2-3-4 trees). This ratio will be small as long as there are not many nodes with only one child.

The only time a node in the computation tree has a single child is if $s = 2$ or $s = 3$. That single child has only one child if $d + 1$ is 2 or 3; but then $s = 1$. From these observations it can be argued that the total number of nodes in the computation tree is at most $4 \cdot |\mathbf{B}(n)|$. Thus the total amount of computation used by the algorithm is $O(|\mathbf{B}(n)|)$; i.e. a constant amount of computation is used to generate each tree sequence, in an amortized sense. Up to constant factors, no algorithm for generating tree sequences can be faster. Note, however, that the number of sequences grows exponentially, so test generation is limited to relatively small values of n (i.e. the enumeration is inherently inefficient, but is being done optimally).

3.3. Generating all red–black trees

To every red–black tree there naturally corresponds a unique 2-3-4 tree. This natural correspondence may be explained by three simple recursive transformation rules as shown in Figure 6. Each triangle represents a subtree and each circle is a node; the subtrees in the red–black tree are assumed to have black roots. Note that each of the four cases for the colours of the two children of the root of a black-rooted red–black tree are represented on the right-hand side of Figure 6. These rules indicate precisely how many red–black trees correspond to each 2-3-4 tree sequence. The number of black-rooted red–black trees corresponding to $a[0\,.\,.p–1]$ is 2^m where m is the number of 3’s in a. In addition, the root of a black-rooted red–black tree can be made red as long as its two children are black; i.e. if $a[p - 1] = 2$. For example, the tree sequences for $n = 6$ are 242, 332, 422 and 2223, and

```
void ttf( int s, int d, int p ) {
   if (s == 1) { PrintIt( p ); return; }
   if (s <= 4) { a[p] = s;  ttf( d+1,  0,  p+1 ); }
   if (s >= 4) { a[p] = 2;  ttf( s-2, d+1, p+1 ); }
   if (s >= 5) { a[p] = 3;  ttf( s-3, d+1, p+1 ); }
   if (s >= 6) { a[p] = 4;  ttf( s-4, d+1, p+1 ); }
} /* ttf */
```

Figure 5. Recursive algorithm for generating all 2-3-4 tree sequences.

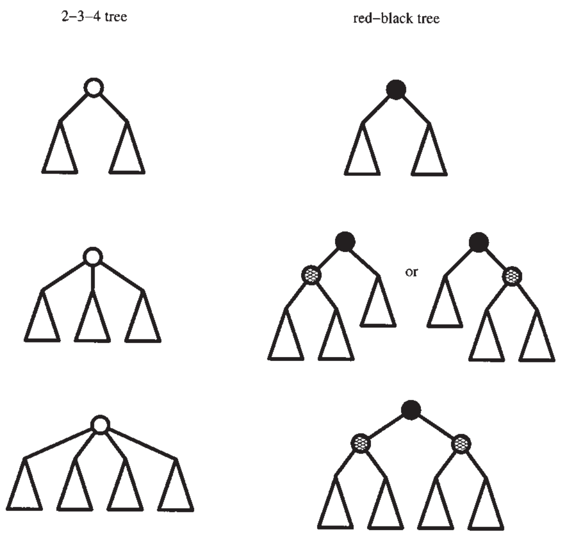


Figure 6. The 2-3-4 tree ⇔ red–black tree conversion rules.

give rise to $1+4+1+2=8$ black-rooted red–black trees, and $2+8+2+2=14$ red–black trees in total.

An algorithm to generate red–black trees is displayed in Figure 7. The overall structure of the algorithm is the same as that of **ttf** in Figure 5. The main difference is that, instead of storing numbers in an array to enumerate the tree, this algorithm creates the subtrees given in Figure 6 as data structures and links them together to form an entire tree.

```
void R( int t, int leaf, int s, int d, int p, RBN *cur ) {
   RBN *node;
   node = CreateSubtree( t );
   if (leaf) AddLeaf( node, t );
   else AddInternal( node, t, &S, &cur );
   Push( &S, node );
   RB( s, d, p, cur );
   Pop( &S );
} /* R */

void RB( int s, int d, int p, RBN *cur ) {
   if (s == 1) { Process( S );  return; }
   if (s <= 4) R( s, p==d, d+1,  0,  p+1, cur );
   if (s == 3) R( 5, p==d, d+1,  0,  p+1, cur );
   if (s >= 4) R( 2, p==d, s-2, d+1, p+1, cur );
   if (s >= 5) R( 3, p==d, s-3, d+1, p+1, cur );
   if (s >= 5) R( 5, p==d, s-3, d+1, p+1, cur );
   if (s >= 6) R( 4, p==d, s-4, d+1, p+1, cur );
} /* RB */
```

Figure 7. An algorithm to generate all red–black trees.

The main body of the algorithm is in procedure **RB**. It is identical with **ttf** except for the following points.

(1) Each occurrence of **a[p]** $= x$; **ttf(**a,b,c**);** is replaced with a call to **R(**x, **p**$==$**d**, a, b, c, **cur);**.
(2) Each occurrence of **a[p]** $=$ **3** is replaced by two calls, one with 3 and one with 5. These handle the two possibilities that occur when transforming a node with 3 children (3 is for the case with the *red* child on the left and 5 is for the red child on the right).
(3) When a red–black tree is printed, a check is made to see if both children of the root are black. If this is the case then a second red–black tree is printed with the root colour changed to *red*.

The initial call is **RB(**$n-1$,0,0,**NULL)** to generate all red–black trees with n internal nodes. The procedure **R** does the actual manipulation of the red–black tree.

(1) It creates the type subtree corresponding to the number of children of the 2-3-4 tree (see the transformation rules in Figure 6).
(2) It links in the children, either previous subtrees or leaves as appropriate, using the procedures **AddInternal** and **AddLeaf**.
(3) It recursively invokes **RB** with the new parameters.
(4) Finally, it resets the tree structure before returning.

As each subtree is created it is pushed onto a stack. It is popped back off once the subtree is no longer needed (after the recursion returns). The size of this stack corresponds to the index into the array (the variable **p**) in Figure 5.

The pointer **cur** indicates the lowest element of the stack that does not currently have a parent. When a new subtree is created and it needs children (non-leaves), it uses the subtrees pointed to by **cur** (without removing them) and then updates **cur** to the next available subtree. This imitates the top-to-bottom, level-by-level, from right-to-left behaviour of the $B(s,d)$ tree algorithm. The parameter **p** is used to determine if children are needed. If, when a new node is created, **p** $==$ **d**, then the subtree must be at the bottom of the tree so its children must all be leaves. Otherwise, the subtree must take its children from those at the **cur** pointer. This is why **p** $==$ **d** is passed as an argument to the **R** procedure.

The only other consideration is that a black-rooted red–black tree with two black children can be converted into another valid red–black tree by making the root red.

3.3.1. Analysing the red–black tree generating algorithm

The analysis of the tree generation algorithm closely follows the reasoning used to analyse **ttf**. The only additional consideration is the overhead in generating the tree structures and not just the tree sequences. The only additional observation to make is that each of the additional procedures, **Push**, **Pop**, **AddInternal** and **AddLeaf** can be implemented so that they take constant time. Thus the red–black tree generation algorithm produces each tree in constant time, in an amortized sense.

3.4. The preorder representation of a red–black tree

Binary trees are often represented by doing a preorder traversal of the tree, recording a 1 for each internal node, and a 0 for each leaf, omitting the final leaf. Red–black trees are a particular type of binary tree, with internal nodes labelled either red or black. A natural alternative to the string representation used previously is to use a preorder labelling, recording a 0 for each leaf, a 1 for each black internal node, and a 2 for each red internal node. See Figure 8 for an example. This representation is useful for storing and restoring the trees, but is not so useful for generating the trees.

4. TEST DRIVERS

This section presents three test drivers: two of black-box type and one white-box type. In all cases, the class-under-test (CUT) is the **`IntSet`** container class (see Figure 2).

4.1. `IntSet`: test case selection

In testing **`IntSet`**, the element values themselves are relatively unimportant while the insertion order is significant. To keep driver development costs low, it is important that it is easy to generate the values *and* to check that the results are correct. Further, for a given CUT state, insertions should be performed before and after every value in the state. All of these requirements are met with states that are instances of the following simple pattern:

$$\{2,4,\ldots, 2n\} \text{ for } n \geq 0$$

There are three drivers, differing in the states that are selected for generation. The ordered black-box (OBB) driver simply adds the set elements in increasing order. The random black-

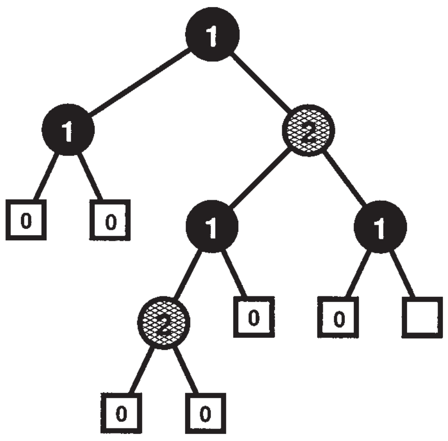


Figure 8. The red–black tree with preorder labelling 110021200010.

box (RBB) driver generates a number of states for each set size by adding the elements in quasi-random order, using *linear congruential sequences* [28]. Given the sequence length $N \geq 0$ and the increment $D \in [1 \,..\, N-1]$:

$$\mathrm{lcs}(N,D) = \langle 0, D \bmod N, 2D \bmod N, \ldots, (N-1)D \bmod N \rangle$$

If N and D are relatively prime, then $\mathrm{lcs}(N,D)$ is guaranteed to be a permutation of $0,1,\ldots,N-1$. For example, $\mathrm{lcs}(5,2) = \langle 0,2,4,1,3 \rangle$. One special case is worth noting:

$$\mathrm{lcs}(N,1) = \langle 0,1,\ldots,N-1 \rangle$$

Unlike the black-box drivers, the white-box (WB) driver is closely tied to the underlying data structure. For a set with n values, it uses the algorithm presented in the previous section to generate every red–black tree with n nodes.

4.2. **`IntSet`: test drivers**

Figure 9 contains pseudocode for the three drivers. Following the strategy in Figure 1 each driver contains a function (**`OBB`**, **`RBB`** or **`WB`**) to generate a sequence of CUT states and a function (**`checkState`**) to check that the CUT behaves properly in that state. The call **`OBB(`**n**`)`** runs $n + 1$ tests, one for each $i \in [0..n]$. For a given i value, an empty **`IntSet`** object is created, loaded with $2,4,\ldots,2i$ with a trivial for-loop, and passed to **`checkState`**. The **`RBB`** driver is similar except, for each $i \in [0..n]$, a linear congruential sequence is generated for each $d \in [1..i-1]$, which is relatively prime with i.

The **`WB`** function is based on the tree generation algorithm of Section 3. While that algorithm generates all red–black trees, the **`IntSet`** implementations tested here produce only trees with black roots [3]. Thus, the **`WB(`**n**`)`** function ignores trees with red roots. The call **`WB(`**n**`)`** runs one test for each $t \in RB(n)$, the set of all black-rooted red–black trees with n nodes. For each tree t, an empty **`IntSet`** object is created, loaded with t using **`setTree`**, and passed to **`checkState`**. Ideally, **`setTree`** would be implemented by a sequence of **`add`** calls, e.g. loading the node values in breadth-first order. What is needed here is an algorithm that takes a red–black tree t as input and produces a sequence of add calls that would generate t in the **`IntSet`** implementation. It turns out that this problem is a difficult one and the known solutions are both complex and sensitive to the particular node insertion algorithm used [29]. The problem is hard even if deletions are permitted. With the red–black tree algorithms tested in this paper and used in the Standard Template Library, many trees cannot be created using add calls alone, and it is not obvious how to generate **`add/remove`** sequences that generate a given tree. Consequently, the **`setTree`** member function was added to **`IntSet`** strictly for testing purposes. The **`setTree`** implementation builds the tree directly, bypassing the tree-balancing algorithms in **`add`** or **`remove`**.

The implementation of **`checkState`** is divided into three parts.

Step 1. *Check the contents of* **`s`** *with the* **`isMember`** *get call*. The contents of **`s`** are verified by calling **`isMember(`**i**`)`** for every $i \in [1 \,..\, 2n+1]$.

```
OBB(int n)
{    for (int i = 0; i <= n; i++) {
          IntSet s;
          for (int j = 1; j < i; j++)
               s.add(2*j);
          checkState(s,i);
     }
}
RBB(int n)
{    for (int i = 0; i <= n; i++)
          for (int d = 1; d < i; d++)
               if (gcd(d,i) == 1) {
                    IntSet s;
                    foreach j $\in$ lcs(i,d)
                         s.add(2*j);
                    checkState(s,i);
               }
}
WB(int n)
{    for each $t \in RB$(n) {
          IntSet s;
          s.setTree($t$);
          checkState(s,n);
     }
}
checkState(IntSet s, int n)
{
     check that s contains $\{2, 4, \ldots, 2n\}$

     for (int i = 1; i <= 2*n+1; i++) {
          IntSet s0(s);
          s.add(i);
          if (i % 2 == 0) check that s contains $\{2, 4, \ldots, 2n\}$
          else check that s contains $\{2, 4, \ldots, 2n\} \cup \{i\}$
     }
     for (int i = 1; i <= 2*n+1; i++) {
          IntSet s0(s);
          s.remove(i);
          if (i % 2 == 0) check that s contains $\{2, 4, \ldots, 2n\} \setminus \{i\}$
          else check that s contains $\{2, 4, \ldots, 2n\}$
     }
}
```

Figure 9. Pseudocode for black-box and white-box drivers.

Step 2. *Perturb* **`s`** *by adding a single element* with the **`add`** set call. For every $i \in [1\,..\,2n+1]$, a copy of **`s`** is made in **`s0`** and i is added to that copy. If i is even then the **`add`** call is ignored. In either case, the contents of **`s0`** are checked, as in step 1.

Step 3. *Perturb* **`s`** *by removing a single element with the* **`remove`** *set call*. For every $i \in [1 \,.\,.\, 2n + 1]$, a copy of **`s`** is made in **`s0`** and i is removed from that copy. If i is odd then the **`remove`** call is ignored. In either case, the contents of **`s0`** are checked, as in step 1.

For simplicity, code that purposely generates an exception, such as adding the same element twice, has been omitted. It would be easy to modify **`checkState`** to perform exception tests using, for example, the methods described by Hoffman and Strooper [19].

5. EXPERIMENTAL EVALUATION

The drivers were applied to 12 **`IntSet`** implementations based on red–black trees, developed by students in a fourth-year software engineering course. Three of the 12 implementations contained faults. Both the white-box driver (WB) and the black-box drivers (OBB and RBB) caused all three faulty implementations to fail, causing addressing exceptions in each case. Careful study of the faulty programs showed that all three faults were ‘transcription errors’: errors in converting the pseudocode in the text [3] to C++. No coverage analysis was performed on the faulty implementations because their tests did not run to completion. The coverage characteristics of one of the nine fault-free implementations is reported later in this section. Statement and path coverage of the remaining eight fault-free implementations showed no significant differences.

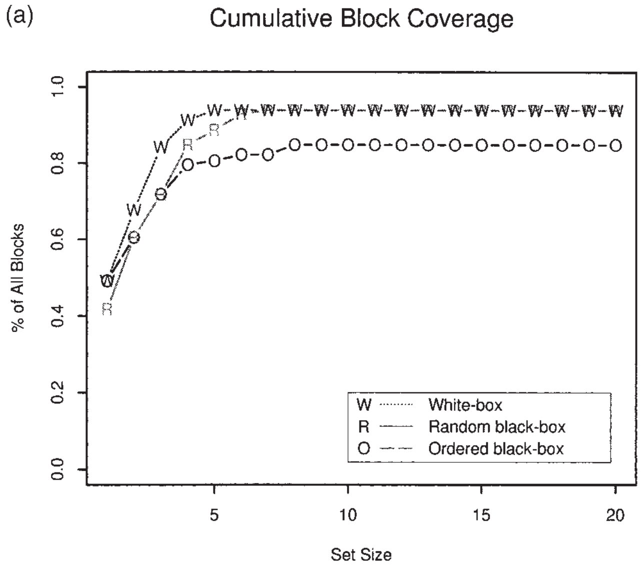
5.1. Driver coverage results

The PP path profiling tool of Ammons, Ball and Larus [30,31] was used to analyse the basic block coverage and path coverage of the fault-free red–black tree implementation under the three drivers. PP instruments Unix executable files for Sparc/Solaris (i.e. **`a.out`** files). PP identifies the basic blocks and control transitions in an executable, creates a control flow graph for each procedure, and inserts small blocks of machine code to record which paths are executed. PP profiles intra-procedural acyclic control flow paths that start either at procedure entry or a loop backedge and end at procedure exit or a loop backedge. From this path coverage information, basic block coverage information can be derived.

Figure 10(a) shows cumulative basic block coverage for all three drivers, while Figure 10(b) shows cumulative path coverage for the three drivers. The *x*-axis represents the size of the integer set used in the test. The *y*-axis represents block/path coverage over all blocks and paths, including unreachable blocks and infeasible paths. For the given red–black tree implementation, the PP tool identified a total of 309 basic blocks and 187 paths.

The graph of Figure 10(a) shows that all three drivers achieve very high block coverage of the red–black tree implementation. In fact, the RBB and WB achieve 100 per cent of all reachable blocks. OBB achieves lower coverage than RBB, as expected. WB achieves coverage faster than OBB and RBB as it examines more trees per test. OBB was run to a set size of 200 and block coverage did not increase over that shown in this graph.

The path coverage graph of Figure 10(b) shows that path coverage distinguishes the three drivers better than block coverage does. Table I breaks down the coverage for RBB and WB



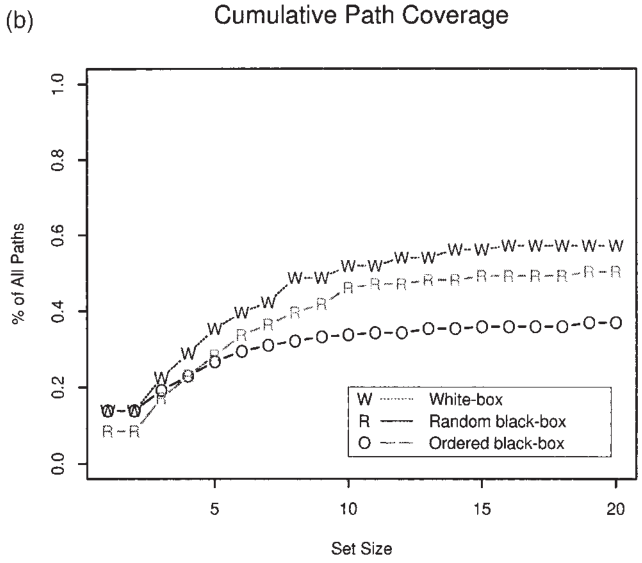


Figure 10. Cumulative block and path coverage for the red–black tree implementation under three different test drivers. The *x*-axis represents the size of the integer set used in the test.

by method. The set of paths covered by WB but not RBB stems primarily from one method: **`RBDeleteFixup`**. RBB was run out to a set size of 200; no new paths were encountered after set size 19. WB was run up to a set size of 20, after which further testing became prohibitively expensive (there are 248 312 red–black trees of size 20).

The **`RBDeleteFixup`** method was analysed to understand why white-box testing was so

Table I. Summary of path coverage by method for the RBB and WB. ‘Total paths’ is the total number of paths in the method, as determined by the PP tool. ‘RBB’ is the number of paths covered by the random black-box driver and ‘WB’ is the number of paths covered by the white-box driver.

Method	Total paths	RBB	WB
`isMember`	1	1	1
`removeAll`	1	1	1
`size`	1	1	1
`IntSet0` (constructor)	2	1	1
`IntSet0` (destructor)	2	1	1
`postorderTreeWalk`	2	2	2
`remove`	2	2	2
`add`	3	2	2
`treeMinimum`	4	4	4
`leftRotate`	6	6	6
`rightRotate`	6	6	6
`treeSuccessor`	7	1	1
`treeSearch`	8	8	8
`treeInsert`	10	7	7
`RBInsert`	16	16	16
`RBDeleteFixup`	44	24	36
`RBDelete`	72	11	12

much more effective in covering its paths than black-box testing. WB covers 36 of 44 possible paths. The remaining paths were examined and all were found to be infeasible. RBB covered only 24 of the 44 paths. Perhaps not surprisingly, this method is the largest of all the methods and has the most complex control flow. It restores the red–black tree properties after the deletion of a node by a fairly complex traversal and restructuring of the tree. By testing this method against all red–black trees of size *N*, WB is able to achieve much more thorough coverage than the RBB. However, as shown in Table I, the two drivers have identical coverage for all other procedures other than **`RBDeleteFixup`** and **`RBDelete`**.

Path coverage is less than block coverage because of infeasible paths, contributed primarily by one method: **`RBDelete`**. This method has 72 paths, but only 12 feasible paths. This accounts for 60 infeasible paths out of the total of 187 potential paths in the implementation. Simple restructuring of this method to eliminate the infeasible paths resulted in a method with 15 paths and 12 feasible paths. The implementation achieves better performance with the restructured routine, as some redundant computations were eliminated.

Of course, path coverage results are hard to use in practice because of infeasible paths. However, several ideas are suggested by the case study.

(1) While the number of infeasible paths is unknown, the rate of increase in the number of paths covered over a series of tests can be used to evaluate whether a particular testing strategy is helpful in covering paths not previously executed. As the two graphs in Figure 10 show, new paths are being covered when basic block coverage has levelled off.

(2) The more thoroughly a test suite exercises a program, the higher the probability that unexecuted paths are infeasible paths. In other words, a thorough test driver provides an approximation to the set of feasible paths in a program. In the example, all the unexecuted paths examined for WB path coverage were infeasible paths.
(3) From the analysis of the `RBDelete` method it was found that the presence of many infeasible paths may make code unnecessarily slow and hard to understand. In some cases, restructuring the code to eliminate infeasible paths also improves both its performance and the ability of programmers to understand and modify it. Other researchers have reported similar relationships between infeasible paths, and performance and comprehensibility [32,33].

5.2. Cost of testing

This section presents timings of the three drivers, as shown in Figure 11. The drivers were run on a Sparc-server E-3000. The *y*-axis shows the amount of time (in seconds of elapsed real time) to run each of the three (PP-instrumented) drivers for each set size up to 20. These timings are not cumulative. Not surprisingly, for larger set sizes, the WB driver takes appreciably longer than the black-box drivers, since the number of trees considered for each set size grows exponentially with set size, while OBB considers 1 set for each set size and RBB considers at most *N* sets for each set size. The WB times are dominated by inserting and deleting every element, for every possible tree. The time taken to generate all the trees is insignificant. For example, it takes only six seconds to generate all red–black trees of

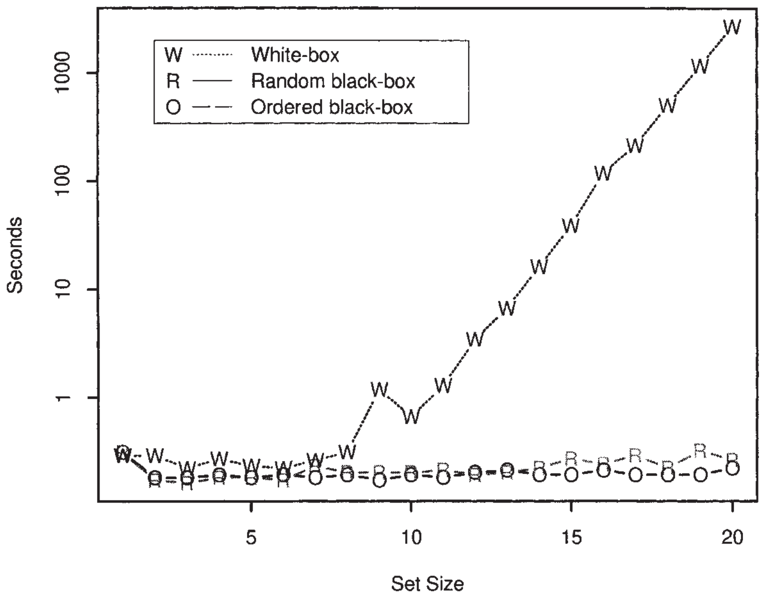


Figure 11. Timings of the three instrumented drivers, in seconds of real time, for each set size. The timings are not cumulative. Also note that the *y*-axis is on a logarithmic scale.

size 20, while it takes 2707 seconds (45 minutes) for WB to run on all red–black trees of size 20.

Despite the longer running time of the WB driver compared to black-box, the times are reasonable, given the higher level of path coverage achieved. It took about 1.5 hours to run WB on all trees up to and including trees of size 20. The path coverage plateau at $N = 19$ suggests stopping there for white-box regression testing. None the less, if one plans to employ white-box testing for a large number of container classes, such test times may be unacceptably long, suggesting a randomized approach to WB testing.

5.3. Randomized white-box testing

This section explores how randomizing the selection of red–black trees can affect the coverage and run-time results for the white-box driver. For this experiment, inputs consisted of all red–black trees ranging in size up to 20 (there are 479 843 such trees). A randomizing filter was applied to this input. The filter passes a tree through if a randomly generated number between 0 and 1 is less than a certain threshold, and throws the tree away otherwise. The WB driver was run using a sequence of ten threshold values, generated as follows: the last value in the sequence is 0.25 per cent (1/4 of one per cent); the *i*th value in the sequence is half of the $(i + 1)$th value. Given that there are 479 843 trees to choose from, this means that approximately two trees will be picked for the first threshold value and approximately 1200 trees will be picked for the last threshold value.

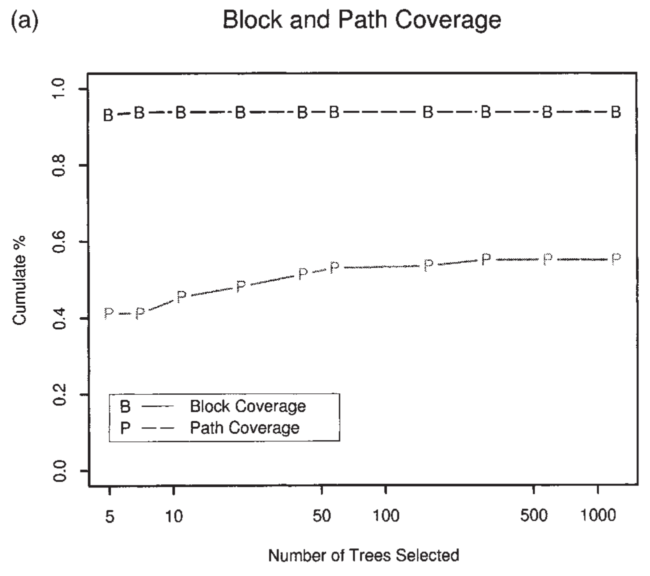
Figure 12 shows the cumulative coverage and (non-cumulative) timing results for the randomized WB driver. In both graphs, the *x*-axis represents the actual number of trees chosen by the filter for the given threshold value. The main point to note from these two plots is that in 12 seconds, the randomized WB driver (RWB) was able to achieve coverage very close to that of the WB driver. In fact, there are only four paths from the WB driver not covered by the RWB driver. Furthermore, even with small threshold values, the path coverage is quite good.

Using randomization, it is also possible to explore trees with much larger numbers of nodes. This would require adapting the red–black tree generation algorithm from its currently batch-oriented flavour to randomly generated trees of a given size in an on-line fashion.

6. CONCLUSIONS

This paper has presented a generic approach to testing container classes, based on state generation. The approach has been demonstrated by developing: (1) a new and efficient algorithm for red–black tree generation; and (2) test drivers for classes implemented with red–black trees. Initial experiments indicate that the approach is effective, producing significantly better path coverage than black-box tests, and fault detection as good as the black-box tests on a sample of faulty implementations. The drivers are also practical in terms of run time and development cost. Test run times were reasonable because the tree generation algorithm and the **`IntSet`** implementation are efficient. Each driver consists of roughly 100 lines of straightforward C++; only the tree generation algorithm is complex.

The approach provides a way for testing practitioners to harvest the prodigious output of



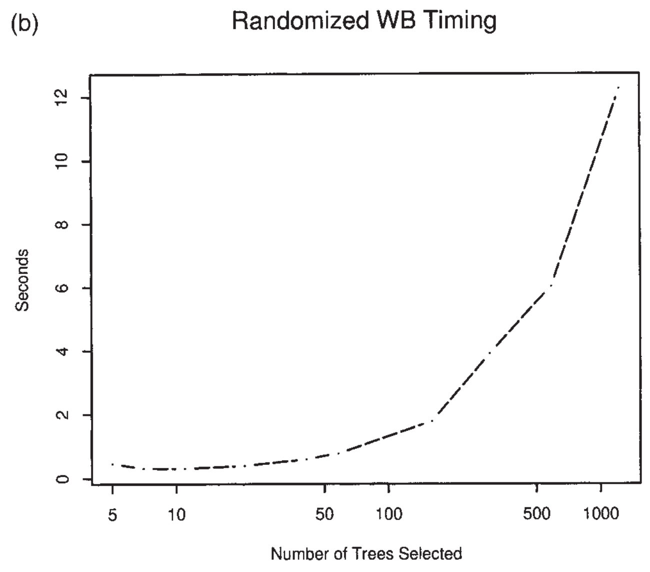


Figure 12. Coverage and timing for the randomized black-box driver.

the theory of algorithms community. While the generation algorithms are complex there are many already available. Further, the tester needs to understand the algorithm output but can ignore its implementation. Since there are not very many data structures in wide use commercially, a modest collection of generation algorithms will be sufficient to support a lot of class testing.

ACKNOWLEDGEMENTS

DH was supported in part by NSERC and Bell Laboratories. FR was supported in part by NSERC. Thanks to Nigel Daley for prototyping the white-box drivers.

REFERENCES

1 McDonald J, Hoffman D, Strooper P. Programmatic testing of the Standard Template Library container classes. In *Proceedings of IEEE International Conference on Automated Software Engineering* 1998; 147–156.
2 JavaSoft. *The bug parade*. http://java.sun.com/jdc [1999].
3 Cormen T, Leiserson C, Rivest R. *Introduction to Algorithms*. MIT Press: Cambridge, MA, 1990
4 Clarke L. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* 1976; **2**(3): 215–222.
5 Bertolino A, Marré M. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering* 1994; **20**(12): 885–899.
6 Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis* 1998; 53–62.
7 Parasoft. *Automatic white-box testing for the Java developer*. http://www.parasoft.com/jtest/jtestwp.htm [1998].
8 Rothermel G, Harrold M. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2): 173–210.
9 Bougé L, Choquet N, Fribourg L, Gaudel M. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software* 1986; **6**(4): 343–360.
10 Gannon J, McMullin P, Hamlet R. Data-abstraction, implementation, specification and testing. *ACM Transactions on Programming Languages and Systems* 1981; **3**(3): 211–223.
11 Hughes M, Stotts D. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis* 1996; 53–61.
12 Chen HY, Tse TH, Chan FT, Chen TY. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1998; **7**(3): 250–295.
13 Dillon L, Ramakrishna Y. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* 1996; 106–117.
14 Stocks P, Carrington D. A framework for specification-based testing. *IEEE Transactions on Software Engineering* 1996; **22**(11): 777–793.
15 Panzl D. A language for specifying software tests. In *Proceedings of the AFIPS National Computer Conference*. AFIPS, 1978; 609–619.
16 Hoffman D. A CASE study in module testing. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society, 1989; 100–105.
17 Hoffman D, Strooper P. Automated module testing in Prolog. *IEEE Transactions on Software Engineering* 1991; **17**(9): 933–942.
18 Murphy G, Townsend P, Wong P. Experiences with cluster and class testing. *Communications of the ACM* 1994; **37**(9): 39–47.
19 Hoffman DM, Strooper PA. Classbench: A framework for automated class testing. *Software-Practice and Experience* 1997; **27**(5): 573–597.
20 Ostrand T, Balcer M. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31**(6): 676–686.
21 Software Research Inc. *The TDGEN test data generator*. http://www.soft.com/products/index.html [1998].
22 Cobb R, Mills H. Engineering software under statistical quality control. *IEEE Software* 1990; **7**(6): 44–54.
23 Musa J, Iannino A, Okumoto K. *Software Reliability Measurement, Prediction, and Application*. McGraw-Hill: New York, 1987.
24 Lucas J, van Baronaigien D, Ruskey F. On rotations and the generation of binary trees. *Journal of Algorithms* 1993; **15**(3): 343–366.
25 Kelsen P. Ranking and unranking trees using regular reductions. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, LNCS #1046. 1996; 581–592.
26 Nijenhuis A, Wilf H. *Combinatorial Algorithms* (2nd edn). Academic Press: New York, 1978.
27 Doong R, Frankl P. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(2): 101–130.
28 Knuth D. *The Art of Computer Programming*, Vol. II. Addison-Wesley: Reading, MA, 1969.
29 Cameron H, Wood D. Insertion reachability, skinny skeletons, and path length in red–black trees. *Information Sciences* 1994; **77**(1–2): 141–152.

30 Ball T, Larus JR. Efficient path profiling. In *Proceedings of MICRO 96*. 1996; 46–57.
31 Ammons G, Ball T, Larus J. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices* 1997; **32**(5): 85–96. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation.
32 Hedley D, Hennell M. The causes and effects of infeasible paths in computer programs. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 1985; 259–266.
33 Woodward M, Hedley D, Hennell M. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering* 1980; **6**(3): 278–286.