State Coverage: Software validation metrics beyond Code Coverage - Extended Version

Dries Vanoverberghe
Jonathan de Halleux
Nikolai Tillmann
Frank Piessens

Report CW 610, October 2011

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

State Coverage: Software validation metrics beyond Code Coverage - Extended Version

Dries Vanoverberghe
Jonathan de Halleux
Nikolai Tillmann
Frank Piessens

Report CW 610, October 2011

Department of Computer Science, K.U.Leuven

Abstract

Currently, testing is still the most important approach to reduce the amount of software defects. Software quality metrics help to prioritize where additional testing is necessary by measuring the quality of the code. Most approaches to estimate whether some unit of code is sufficiently tested are based on code coverage, which measures what code fragments are exercised by the test suite. Unfortunately, code coverage does not measure to what extent the test suite checks the intended functionality.

We propose *state coverage*, a metric that measures the ratio of state updates that are read by assertions with respect to the total number of state updates, and we present efficient algorithms to measure state coverage. Like code coverage, state coverage is simple to understand and we show that it is effective to measure and easy to aggregate. During a preliminary evaluation on several open-source libraries, state coverage helped to identify multiple unchecked properties and detect several bugs.

State Coverage: Software validation metrics beyond Code Coverage - Extended Version

Dries Vanoverberghe[1]*, Jonathan de Halleux[2], Nikolai Tillmann[2], and Frank Piessens[1]

[1] Katholieke Universiteit Leuven, Leuven, Belgium
{dries.vanoverberghe,frank.piessens}@cs.kuleuven.be
[2] Microsoft Research, Redmond, WA, USA
{jhalleux,nikolait}@microsoft.com

Abstract. Currently, testing is still the most important approach to reduce the amount of software defects. Software quality metrics help to prioritize where additional testing is necessary by measuring the quality of the code. Most approaches to estimate whether some unit of code is sufficiently tested are based on code coverage, which measures what code fragments are exercised by the test suite. Unfortunately, code coverage does not measure to what extent the test suite checks the intended functionality.
We propose *state coverage*, a metric that measures the ratio of state updates that are read by assertions with respect to the total number of state updates, and we present efficient algorithms to measure state coverage. Like code coverage, state coverage is simple to understand and we show that it is effective to measure and easy to aggregate. During a preliminary evaluation on several open-source libraries, state coverage helped to identify multiple unchecked properties and detect several bugs.

Key words: state coverage, test adequacy metric, test oracle

1 Introduction

As software becomes a central part of society, the impact of software defects on the economy is huge. For example, in 2002, software failures were estimated to cost the US economy about $60 billion annually [18]. Currently, testing is still the most important approach to reduce the amount of software defects.

During the testing process, the code under test is exercised in various ways while a test oracle (e.g. assertions or pre- and post conditions) checks that the code behaves according to its specification. Defects are reported and fixed and the testing process restarts. In principle, this process can continue forever since testing usually cannot show the absence of software defects. In practice however, only limited resources are available and testing needs to stop at some point. Software quality metrics help to prioritize where additional testing is necessary by measuring the quality of the code under test.

* This work was done during an internship at Microsoft Research. Dries Vanoverberghe is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the IWT, and by the Research Fund K.U.Leuven.

Currently, most *software validation metrics*, i.e. metrics that estimate whether the code is sufficiently tested, are based on code coverage. Code coverage estimates the fraction of the execution paths of the code under test that are exercised by the test suite. Since code coverage metrics are simple to understand and efficient to compute, the use of code coverage metrics during the testing process is well-established. Furthermore, automatic tools have been created to help testers achieve high code coverage (e.g. random testing, symbolic execution[13]). Unfortunately code coverage alone is not sufficient to measure software quality since it only measures whether the code has been sufficiently exercised. It does not measure the strength of the test oracle, the properties that must be satisfied by the code.

In this paper, we focus on the use of assertions, one of the most basic ways to instrument the code with the test oracle. Whenever the execution reaches an assertion, the execution state must satisfy the given boolean expression. Although the use of assertions is far from new [10, 25, 12, 21] and experimental evidence [15] shows there is a correlation between the number of assertions and the amount of software defects, little work has been done to measure the quality of assertions in a test suite.

We propose the use of *state coverage* [14], a software validation metric based on the hypothesis that every update to the execution state must eventually be followed by an assertion that reads the updated value. State coverage is orthogonal to code coverage: they measure different concerns. While state coverage measures the strength of the test oracle, code coverage measures how well the code is exercised. Nonetheless they are intertwined, for example adding extra assertions to the test suite may decrease code coverage and exercising more paths of the program may discover new state updates and decrease state coverage. Therefore code coverage and state coverage work best in combination. In addition, the thought process of developers to achieve high state or code coverage is also orthogonal: While code coverage makes a developer think in terms of branches, state coverage makes a developer think in terms of properties that are established by state updates.

For a good software validation metric, the following criteria are essential:

- **easy to understand**, for developers and testers who write code and tests to achieve certain metric numbers, and for managers to decide when a project is ready to be shipped,
- **composable**, i.e. results from individual test cases can be combined to an overall result for an entire test suite,
- **effective to measure**, i.e. adding only a reasonable overhead during the software development and testing process.

We show in this paper that state coverage fulfills all of the above criteria.

Except for Mutation Testing [8], state coverage is the only technique to measure the quality of the test oracle. Unfortunately, the mutation adequacy score is hard to understand because deciding whether a live mutant is equivalent can be complex and often requires human intervention. In addition, it suffers from a high performance penalty caused by executing the test suite with millions of mutants.

We have implemented a prototype of the state coverage metric for the .NET platform, and have applied it to several open-source libraries. While adding extra assertions to increase state coverage, we have found several bugs in DSA[2, 1], a library with complementary data structures for the .NET platform. In total, we found seven properties which were not or insufficiently checked in the existing test suite.

To summarize, the main contributions of this paper are:

– We propose a general definition for *state coverage*, a software validation metric that goes beyond code coverage. Our definition improves on existing work by Koster et al.[14] by allowing more dynamic state updates and lifting the restriction on the structure of test cases.
– We present efficient algorithms to measure *object sensitive* and *object insensitive* state coverage, two variants with different granularity.
– We propose a technique to make *object sensitive* state coverage composable.
– We evaluate the metric in a case study on several open-source libraries, using a prototype implementation of our algorithm.

The remainder of this paper is structured as follows. First, Section 2 motivates why additional software validation metrics are desirable using a simple example. Next, Section 3 introduces state coverage and discusses how it can be computed. Then, we evaluate state coverage in Section 4. Finally, we discuss related work and conclude in Sections 5 and 6 respectively.

2 Motivating example for state coverage

Figure 1 shows the code for a doubly linked list. The method *Append* adds a new value at the end of the list, and the method *Contains* checks whether the list contains a particular value. In addition, the method *TestAppend* is a small test case that adds a few values to the list and checks whether the list contains the added values. The test case executes without failing assertions, and reaches 100% basic block coverage.

Nonetheless, the code contains a bug: A careful inspection of line 11 in the method *Append* reveals that the previous pointer of the new node is not correctly updated when appending the second element to the list. Because the method *Contains* uses the next pointers to traverse the list, the test case cannot detect this problem. A state coverage tool would report that the *previous* field was updated but never read in an assertion. This feedback motivates the tester to assert an additional property that mentions the *previous* field, for example that the bidirectional associations between the nodes are consistent.

We have translated this example from C# to Java, and used Javalanche [23], one of the most advanced mutation testing tools for Java to assess the quality of the tests. Javalanche created 9 mutants of which one mutant was equivalent and all other mutants were killed, which gives the false impression that this code is well-tested. Just like MuJava [16], Javalanche uses selective mutation which only applies a subset of all known mutation operators because it is more efficient. Unfortunately, none of these operators mutate line 11, which can cause a live, non-equivalent mutant.

3 State coverage

In this section, we propose state coverage, an approach that measures the percentage of the state updates that are verified by an assertion. We start with its definition, and

```
public class DoublyLinkedList {
  public Node head = null;
  public void Append(int value) {
    if (head == null) head = new Node(value);
    else {
      var newNode = new Node(value);
      var tail = head.previous;
      newNode.next = head;
      head.previous = newNode;
      tail.next = newNode;
      newNode.previous = tail.previous;
      //CORRECTED: newNode.previous = tail;
    }
  }
  public bool Contains(int value) {
    if (head != null) {
      var current = head;
      do {
        if (current.value == value) return true;
        current = current.next;
      } while (current != head);
    }
    return false;
  }
  public class Node {
    public Node next, previous;
    public int value;
    public Node(int value) {
      this.value = value;
      previous = next = this;
    }
  }
  public void TestAppend() {
    var list = new DoublyLinkedList();
    list.Append(0); list.Append(1);
    Assert.IsTrue(list.Contains(0));
    Assert.IsTrue(list.Contains(1));
    Assert.IsFalse(list.Contains(2));
  }
}
```

Fig. 1: Doubly linked list

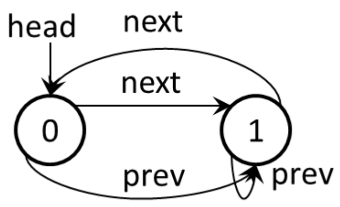

Fig. 2: Doubly linked list nodes

then give a simple algorithm to track state coverage of a single test case at runtime. We describe how state coverage data of individual test cases can be combined into overall state coverage information, in order to measure state coverage of an entire test suite. Finally, we extend the algorithm with dependency tracking to avoid low state coverage ratios due to intermediate state updates and missing context information.

3.1 Definition

We define *state coverage* as the ratio of the number of *state updates* which are read by assertions to the total number of state updates.

This definition of state coverage depends on the definition of state updates. Just as there are different characterizations of code coverage (statement, basic block, arc, etc.), there are different possible characterizations of state coverage, depending on the chosen granularity of state updates.

In this work, we propose two such granularities of state coverage:

- *Object insensitive* state coverage considers as a state update the *code location* in the source code where an update is performed.
- *Object sensitive* state coverage considers as a state update a pair of *object identifier and code location*, where the object identifier is derived from the actual object reference that is involved in a state update at runtime.

Object insensitive state coverage is quite similar in nature to the idea of statement coverage. It simply relates a number of covered code locations to a total number of code locations. While easy to understand, statement coverage is often not fine-grained enough to give confidence that the code has been sufficiently exercised. Similarly, object insensitive state coverage is rather coarse. While it provides some basic insights into the quality of a test suite, we have found cases where only striving for object sensitive state coverage could uncover certain software defects.

We have implemented a prototype to compute state coverage based on runtime monitors (See Figure 3 in appendix). To get the state coverage, all test cases of a given test suite are executed with a special monitor, which gets callbacks during the execution, for example, whenever a field is read or written. Sections 3.2 and 3.3 discuss the implementation of the monitors for object insensitive and object sensitive state coverage. Both monitors collect a set of state updates (*writes*) and a subset that is read in assertions (*reads*). The resulting value is computed by dividing the number of reads by the number of writes.

3.2 Object Insensitive State Coverage

Figure 4a shows the basic algorithm to compute the object insensitive state coverage metric. The class *ObjectInsensitiveStateCoverageMonitor* is a runtime monitor which gets notified whenever a field is written (*WriteField*) or read (*ReadField*) and upon entering (*EnterMethod*) and leaving methods (*LeaveMethod*). The state coverage monitor uses *EnterMethod* and *LeaveMethod* to track whether the execution is currently inside the assert method. Whenever a field is written, the current code location is added to

```
double GetStateCoverage<T>(
  Monitor<T> m, Set<TestCase> testsuite) {
  foreach(var test in testsuite) {
    m.Execute(test);
  }
  int updates = m.writes.Count;
  int covered = m.reads.Count;
  return ((double)covered)/updates;
}
abstract class Monitor<T> {
  Set<T> writes, reads;
  ...
}
```

Fig. 3: Basic algorithm to compute state coverage

a set *writes*, which tracks all code locations where write operations are performed by the program. A code location represents a method and the offset of an instruction in the body of that method. In addition, the written object and field is associated with the current code location in the map *lastWrite*, which tracks the last location where each object field has been written. When the execution is inside an assert method and the execution reads an object field, the last write location for that object field is added to the set *reads*.

For simplicity, the presented algorithm only deals with writes to object fields. However, other parts of the state, such as static fields, array elements, or struct fields, can be handled similarly.

In general, computing the set of object fields that influence an assertion corresponds to information flow analysis [22]. For simplicity, the algorithms in this paper simply track all fields read during the computation of the assertion. However, our implementation uses runtime information flow monitoring to give more precise results.

Composing the results from all test cases of an entire test suite is easy: It simply amounts to computing the union of the respective reads and writes sets.

3.3 Object Sensitive State Coverage

The basic idea of the object sensitive algorithm is similar to the object insensitive algorithm. The main difference is that instead of just tracking code locations for the reads and writes sets, we will track more information, allowing us to distinguish writes which might have happened at the same code location, but were performed on different objects.

The obvious and most general approach would be to track pairs representing the actual object reference together with the code location where it was written. However, this is impractical. While it may allow computing very precise state coverage for individual test cases, joining the information of individual test cases to obtain an overall state coverage ratio for a test suite becomes challenging: It is not clear how to relate the actual object references of different test runs.

Composing state coverage information is important. It is quite common that multiple unit tests check different properties of the same code unit. Some people even consider it bad practice to write multiple assertions in one test case [19]. For example, the

```
class ObjectInsensitiveStateCoverageMonitor
extends Monitor<CodeLocation> {
 Map<Pair<object,Field>,CodeLocation> lastWrite;
 bool InAssert;
 ...
 void WriteField(object o, Field f) {
   var loc = CurrentCodeLocation();
   lastWrite[new Pair(o, f)] = loc;
   writes.Add(loc);
 }
 void ReadField(object o, Field f) {
  if(inAssert) {
     var loc = lastWrite[new Pair(o, f)];
    reads.Add(loc);
  }
 }
 void EnterMethod(Method method) {
  if(method == Assert.IsTrue)
   inAssert = true;
 }
 void LeaveMethod(Method method) {
  if(method == Assert.IsTrue)
   inAssert = false;
 }
}
class Assert {
 static void IsTrue(AssertExpr expr) {
  if(!expr())
   throw new AssertionViolationException();
 }
 delegate bool AssertExpr();
}
```

(a) Object Insensitive State Coverage Monitor

```
class Frame {
 int idCounter;
 Map<object, int> ids;
 int GetLocalId(object o) {
  if(!ids.ContainsKey(o))
   ids[o]= idCounter++;
  return ids[o];
 }
}
class ObjectSensitiveStateCoverageMonitor
extends Monitor<Pair<int,CodeLocation>> {
 Map<Pair<object,Field>,
  Set<Pair<int,CodeLocation>>> lastWrites;
 bool InAssert;
 ...
 void WriteField(object o, Field f) {
  var s = new Set<Pair<int,CodeLocation>>();
  foreach(var frame in CurrentFrames) {
    var loc = frame.CodeLocation();
    var id = frame.GetLocalId(o)
    var p = new Pair(id, loc);
    s.Add(p);
    writes.Add(p);
   }
   lastWrites[new Pair(o, f)] = s;
 }
 void ReadField(object o, Field f) {
  if(inAssert) {
     var ps = lastWrites[new Pair(o, f)];
     reads.AddRange(ps);
  }
 }
}
```

(b) Object Sensitive State Coverage Monitor

Fig. 4: State coverage monitors

methods *PairTest.Test1* and *PairTest.Test2* in Figure 5 check that the constructor correctly initialized the field *x* and *y* respectively. Since the allocated pair *p* will not be identical during the execution of both test cases, a simple union of the read and written fields leads to a joined state coverage of only 50 percent. On the other hand, it also happens that one single test checks multiple properties on different objects (For example, method *PairTest2.Test* in Figure 5). Both test fixtures check full functional correctness for this pair data structure, and yet their state coverage is only 50 percent.

As discussed above, the global nature of object references poses a challenge to join the results of multiple executions of a particular method or for all methods of a particular type. To enable joining for object sensitive state coverage, the state coverage monitor needs to maintain context-insensitive object identifiers. Figure 4b contains an updated version of the object insensitive algorithm. Each frame maintains a map from the actual object references to frame-local object identifiers. Whenever an object identifier is requested and the frame does not yet have an identifier, it allocates the next identifier, represented by returning the *idCounter* and increasing its value by one. All reads and writes use the frame-local identifier instead of the actual object reference. In addition,

```
class Pair {
  int x, y;
  public Pair(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class PairTest {
  void Test1() {
    var p = new Pair(27, 33);
    Assert.IsTrue(p.x == 27);
  }
  void Test2() {
    var p = new Pair(27, 33);
    Assert.IsTrue(p.y == 33);
  }
}
```

```
class PairTest2 {
  void Test() {
    var p = new Pair(27, 33);
    Assert.IsTrue(() => p.x == 27);
    p = new Pair(27, 33);
    Assert.IsTrue(() => p.y == 33);
  }
}
```

Fig. 5: Example to illustrate composition

the last write also tracks the frame-local identifier. For brevity, we omitted the code to push and pop frames when entering and exiting a method.

The resulting set of reads and writes can be unioned together over multiple tests, which makes it possible to report state coverage values for all tests in a particular test fixture or test assembly.

There are alternative approaches to to assign context-insensitive object identifiers. We have chosen the current approach, because it is simple and effective without sacti-ficing precision.

4 Evaluation

We have implemented a prototype for the state coverage metric as an extension for Pex [26], an automatic test input generation tool for .NET developed at Microsoft Research, based on the idea of dynamic symbolic execution: During the execution of a program, Pex maintains a symbolic state and uses it to generate new inputs that drive the execution to some unexplored path of the program. We implemented our runtime monitor on top of Pex, leveraging its extensible instrumentation framework. As shown in the algorithms in Section 3.2 and Section 3.3, the prototype focuses on object field updates.

The aim of this paper is to present and investigate the notion of state coverage. We want to avoid the influence of automatic test generation on the evaluation of state coverage. Therefore, we use existing projects with manually created test suites, and we do not generate additional test cases with Pex. Since state coverage is computed using a run-time monitor, it is essential that these projects already have a test suite with high code coverage. In addition, state coverage is only useful when the test cases use assertions to specify the test oracle. Based on these criteria, we applied our prototype on the following open-source libraries:

– Quickgraph [7] is a managed C# port of the Boost Graph Library.

– Data Structures and Algorithms (DSA) [2, 1] features implementations of data structures and algorithms that complement the data structures in the .NET 3.5 base class libraries.

Ultimately, the main research question is whether code bases with low state coverage are more likely to have bugs, while code bases with high state coverage are unlikely to have bugs. Unfortunately, answering this question is troubled because of two reasons: First, all projects have been tested reasonably well which implies that the likelihood of findings bugs is low. By consequence, the evaluation is biased. Second, we do not have historical information about older bugs. Therefore it is not possible to compute a reliable correlation between the amount of bugs and the state coverage values.

Since a quantitative analysis is challenging, we perform a more qualitative analysis. Our experiment answers the following research question: "for code bases with good structural coverage, how does an increase in state coverage impact the number of bugs found?"

We measure the initial state coverage of each project and manually add new assertions to read object fields that were written but not read in an assertion. When we can no longer improve the state coverage score, we report the amount of added assertions and the number of bugs we discovered. In this process, we give preference to the simplest assertions that increase state coverage over well-known more complicated invariants. In a realistic test setting, more time could be spent to come up with more valuable invariants. This implies that the results are conservative, i.e. it is in some sense the weakest set of invariants that maximizes state coverage.

For one of the projects, DSA, we perform a detailed analysis of the added properties. We discuss the most useful invariants, and assess whether they add value to the test suite. For this project, we also evaluate the level of false positives/negatives. False positives show up as uncovered state updates, and are therefore easy to detect. False negatives are harder to detect. Therefore, we manually inspected the code bases to find patterns that can cause false negatives.

Most of the assertions are added as invariants or post-conditions using Code Contracts [9]. By consequence, the impact on the existing code base is minimal and the added properties are checked at multiple locations.

4.1 General results

Table 1 contains the results of executing our prototype on the original unmodified projects. Since all projects have been reasonably well tested, they have high basic block coverage (column 2). Columns three and four report the object insensitive and object sensitive state coverage. All projects have a high score on the object insensitive state coverage, and therefore require few additional properties to reach the maximum ratio. This is not surprising since they had high code coverage and a significant number of assertions (See Table 2). In fact this represents a significant (conservative) bias of our evaluation. We expect that object insensitive state coverage is more useful on average projects. The object sensitive state coverage ratios are lower, and highlight the need for some useful properties. Table 1 show the results for QuickGraph and DSA after adding additional assertions. For both projects, we achieved the maximal ratio.

Column five and six show the performance overhead of object insensitive and object sensitive state coverage. However, we made no attempt to reduce the execution overhead of the prototype, therefore there may be room to improve these numbers. What is essential about the overhead is that it clearly is just a constant factor off the original performance, not unlike what one would expect from measuring code coverage overhead.

One may object that writing additional assertions just to increase a new metric is a burden for the developer. Column 3 in Table 2, shows the number of lines of code that the traditional test suites required in order to achieve high code coverage. The code added to increase state coverage was negligible compared to the size of the existing test suite.

		State coverage		Performance overhead	
Project	basic block coverage	Obj. insens.	Obj. sens.	Obj. insens.	Obj. sens.
DSA (before)	1580/1608 (98.26%)	69/71 (97.18%)	552/805 (68.57%)	22.89%	29.92%
QuickGraph (before)	553/658 (84.04%)	17/19 (89.47%)	1006/1307 (76.97%)	374.91%	291.17%
DSA (after)	1973/2074 (95.13%)	71/71 (100.00%)	801/801 (100.00%)	45.33%	57.65%
QuickGraph (after)	673/779 (86.39%)	19/19 (100.00%)	1304/1304 (100.00%)	353.93%	276.60%

Table 1: State coverage results before and after adding extra assertions

	Assertions	LOC	
Project	(Added/Total)	(Original/Total)	Bugs
DSA	33/461	999/1036	5
QuickGraph	22/56	373/426	0

Table 2: Added assertions

4.2 Detailed evaluation of DSA

First, we evaluate the bugs we found in DSA while adding properties, and we describe the process that led us to them. Figure 6 (in appendix) contains the relevant fragment of *BinarySearchTree*. First, some locations in the code write to the root field of the binary search tree, and the left and right fields of the nodes, but these writes were never read in an assert. The simplest invariant for reference fields is checking whether they are non-null. Since the root, left and right fields can be null, we needed a more complicated invariant. The simplest invariant we could find was that the amount of nodes in the tree must equal the count field of the BinarySearchTree. After inserting this invariant, one of the existing tests failed. Upon closer inspection, it revealed that a value was ignored when it was already in the tree, but the count was still increased.

Interestingly, when increasing state coverage, a developer thinks differently about the code than when trying to increase traditional code coverage. In code coverage, when a statement is uncovered, a developer needs to look at the *branch condition* that preceeds this code fragment. In state coverage, when a state update is uncovered, a developer is forced to think about the properties that are established by this state update.

This mindset helped us discover the other four bugs. After reaching full object insensitive and object sensitive state coverage, it was surprising that we did not need to specify some properties, in particular, that the reachable nodes in a linked list can not be shared between different linked lists. The existing test suite did not check this property, and both *SinglyLinkedList* and *DoublyLinkedList* have the methods *AddBefore* and *AddAfter* which insert a value before or after a given node. None of the existing tests attempted to invoke these methods with a node that was not in the linked list. The implementation of these methods does not validate that the nodes are in the linked list, and throws a null reference exception or invalidates the invariants concerning the structure of the linked list (For example, See Figure 7 in appendix).

```
class BinarySearchTree {
  ...
  void Add(T item) {
    if (this.Root == null) {
      this.Root = new BinaryTreeNode<T>(item);
    else
      InsertNode(item);
    this.Count++;
  }
  void InsertNode(T value) {
    BinaryTreeNode<T> current = this.Root;
    while (true) {
      if (Compare.IsLessThan(value, current.Value, Comparer)) {
        ... //code to insert left
      } else if (Compare.IsGreaterThan(value, current.Value, Comparer)) {
        ... //code to insert right
      } else {
        // The value to insert is already present, we simply return
        return;
      }
    }
  }
}
```

Fig. 6: Fragment of BinarySearchTree

```
void AddAfter(
  DoublyLinkedListNode<T> node, T value) {
  ValidateAddArgs(node);
  var n = new DoublyLinkedListNode<T>(value);
  if (node == m_tail) {
    n.Previous = m_tail; m_tail.Next = n;
    m_tail = n;
  } else {
    n.Next = node.Next;n.Next.Previous = n;
    node.Next = n;n.Previous = node;
  }
  Count++;
}
```

Fig. 7: Fragment of DoublyLinkedList

The process to find the bugs in the linked lists illustrates that it is not required to have a full functional specification in order to reach full state coverage. This is due to the fact that state coverage is fundamentally an underapproximative measure for the strength of the test oracle. For example, it was also not necessary to check the consistency between previous and next fields of doubly linked list, or sortedness of the values of the binary search tree.

Nonetheless, 33 additional assertions were added to achieve full state coverage. Four of those assertions were introduced in the new test cases that detect the bug in the linked lists. Eleven of the assertions were trivial data structure invariants, which were enforced locally (for example, in the constructor). Those invariant are less useful, but it is likely possible to infer them automatically using existing invariant generation techniques. Now we discuss the most useful properties that were inserted (using 18 out of the 33 assertions):

- We added a post condition that checks if an element that added to a data structure is contained by the data structure (1 post-condition in CollectionBase, 2 more for specialized methods in Deque, a double-ended queue).
- We added an invariant that checks whether the height of all reachable nodes in an AvlTree is consistent with its actual height (1 invariant, 2 post conditions).
- An invariant in BinaryTree checks that the amount of reachable nodes equals the count of the binary tree (1 invariant).
- None of the tests checked that constructors of heap correctly initialize the strategy field (1 postcondition for both constructors).
- Some data structures are a wrapper around other data structure, but do not always check that the Count field is consistent (2 invariants).
- In the linked lists, the tail pointer is null if and only if the head pointer is null. In addition, the next field of the tail and the previous field of the head must be null (5 invariants in total).
- An invariant in the linked lists checks that the amount of reachable nodes equals the count of the binary tree. In addition, it checks that the previous field of all reachable nodes (except the head node) is not null (2 invariants).

In the end, we achieve 100% state coverage, which implies that the potential false positives due to the dependency tracker did not occur in practice. Intuitively, the lack of false positives due to the intraprocedural algorithm can be explained because the results of a method call are usually consumed in the same branch as the invocation. In addition, the false positives due to tracking dependencies at runtime do not occur because the expressions in assertions are typically simple. Finally, we manually examined the source code for patterns that cause false negatives, and we did not find such patterns.

5 Related work

In the broader sense, state coverage is part of the larger area of software quality metrics. Empirical studies have shown that complexity metrics (e.g. Cyclomatic complexity[17]) and object oriented design metrics (e.g. Coupling [5]) can be used to predict defect density (See Catal et al.[3] for a survey on defect prediction). However, such metrics only indirectly help to reduce the defect rate by measuring the quality of the design.

More narrowly, state coverage is a *test adequacy metric* (See Zhu et al.[27] for a survey on test adequacy criteria), it directly measures how well the software has been validated. Structural coverage metrics (such as statement coverage) are most popular in this area. They all measure to some extent which subset of the execution paths of the program are exercised by the test suite. State coverage is orthogonal to these metrics, since it measures the strength of the test oracle. Therefore, state coverage is most powerful in combination with the existing approaches.

State coverage is most closely related with all-defs [20] coverage. The critical difference between both is that dataflow coverage works with all state reads, whereas state coverage focuses on state reads that influence the result of an assertion. This difference makes state coverage measure the strength of the assertions in a code base, instead of measuring whether the code base is sufficiently exercised.

With respect to Koster et al.[14], our definition of state coverage is more general. We do not require any particular structure for the tests. Furthermore, we allow more dynamic state update identifiers (e.g. by including object identifiers) than nodes in the control flow graph. Our preliminary experiments have shown that a more dynamic version can reveal more faults and is therefore more precise. Finally, we do not restrict the metric to output-defining nodes. Therefore our algorithm gives a more accurate view of the fraction of state updates that have been checked by assertions.

Structural test adequacy criteria have also been applied to specifications instead of programs [4, 11, 6]. These metrics consider the system as a black box, and evaluate whether the test suite sufficiently exercises a model of the system. Therefore, they measure the quality of a set of tests rather than the quality of the test oracle. Unlike pure program based or specification based adequacy metrics, state coverage uses the structure of the program and the specification.

Next, fault-based test adequacy criteria (mostly mutation testing [8]) measure the fault finding capability of a test suite. Unlike existing structural test adequacy criteria, mutation testing can be used to evaluate the strength of the test oracle. Mutation testing injects faults into the codebase and checks whether the test suite can observe the injected fault. Often mutation testing requires generating and executing millions of mutants. The mutation adequacy score divides the amount of killed mutants by the amount of non-equivalent mutants. Unfortunately, deciding whether a mutation is equivalent is undecidable in general, and therefore often requires human interaction. State coverage achieves some of the benefits of mutation testing, without the performance overhead and complexity of mutation testing.

The number of assertions in a code base have been shown to correlate inversely with the amount of defects in the code [15]. Based on this observation, counting the number of assertions is an obvious metric for the strength of the oracle. State coverage goes beyond assertion count in that it is more constructive: it highlights parts of the state that are not mentioned by any assertion. In addition, assertion count does not normalize as well as state coverage: The ideal assertion density (the assertion count divided by the size of the code base) may vary from program to program.

From a more technical perspective, state coverage is related to UnitPlus [24], a tool to assist developers in writing unit tests. Based on a static read/write analysis, UnitPlus suggest new assertions. Unlike UnitPlus, our algorithm computes state coverage at runtime and therefore it can be more precise (e.g. we don't have problems due to aliasing). In addition, the algorithm in Section 3 uses dependency tracking to precisely track which state has been read while constructing expressions.

6 Conclusion and future work

In this paper, we went beyond traditional code coverage metrics to assess the quality of a test suite. We created state coverage, a novel metric that measures the ratio of state updates that are read by assertions, and we presented efficient algorithms to measure state coverage. We have implemented a prototype to measure state coverage, and evaluated the metric in a case study on several open-source libraries. State coverage helped to identify multiple unchecked properties and detect several defects.

In future work, we will further experiment with different frame-local object identifiers. Although the current identifiers are simple and efficient, they do not provide enough feedback to debug which object is updated and cover this update in a new assertion.

In addition, we plan to measure some notion of redundancy of assertions to avoid trivial assertions such as tautologies or otherwise implied properties. We envision a redundancy notion which gives individual assertions a score, which measures its quality, similar to how state coverage quantifies the quality of a test case or suite, but possibly another orthogonal metric.

References

1. G. Barnett and L. Del Tongo. *Data Structures and Algorithms: Annotated Reference with Examples*. .NETSlackers, 2008.
2. G. Barnett and L. Del Tongo. Data structures and algorithms (dsa). http://dsa.codeplex.com/, 2008.
3. C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009.
4. J. Chang, D. J. Richardson, and S. Sankar. Structural specification-based testing with adl. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 62–70, New York, NY, USA, 1996. ACM.
5. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
6. F. Dadeau, Y. Ledru, and L. du Bousquet. Measuring a java test suite coverage using jml specifications. *Electronic Notes in Theoretical Computer Science*, 190(2):21 – 32, 2007. Proceedings of the Third Workshop on Model Based Testing (MBT 2007).
7. J. de Halleux. Quickgraph: A 100% c# graph library with graphviz support. http://www.codeproject.com/KB/miscctrl/quickgraph.aspx, 2007.
8. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

9. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, New York, NY, USA, 2010. ACM.
10. R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.
11. M. P. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:178–186, 2004.
12. C. A. R. Hoare. Assertions: A personal perspective. *IEEE Ann. Hist. Comput.*, 25(2):14–25, 2003.
13. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
14. K. Koster and D. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 541–544, New York, NY, USA, 2007. ACM.
15. G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 204–212, Washington, DC, USA, 2006. IEEE Computer Society.
16. Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.
17. T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.
18. N. I. of Standards and technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.
19. R. Osherove. *The Art of Unit Testing with examples in .NET.* Manning Publications Co., 2009.
20. S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11:367–375, April 1985.
21. D. Rosenblum. A practical approach to programming with assertions. *Software Engineering, IEEE Transactions on*, 21(1):19 –31, jan. 1995.
22. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.
23. D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297–298, New York, NY, USA, 2009. ACM.
24. Y. Song, S. Thummalapenta, and T. Xie. Unitplus: assisting developer testing in eclipse. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 26–30, New York, NY, USA, 2007. ACM.
25. R. N. Taylor. Assertions in programming languages. *SIGPLAN Not.*, 15(1):105–114, 1980.
26. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.
27. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29:366–427, December 1997.