A Passive Test Oracle Using a Component's API

Rakesh Shukla, David Carrington and Paul Strooper
School of Information Technology and Electrical Engineering,
The University of Queensland, St. Lucia 4072, Australia.
{shukla, davec, pstroop} @itee.uq.edu.au

Abstract

A test oracle is a mechanism that is used during testing to determine whether a software component behaves correctly or not. The test oracle problem is widely acknowledged in the software testing literature and many methods for test oracle development have been proposed. Most of these methods use specifications or other resources to develop test oracles. A passive test oracle checks the behaviour of the component, but does not reproduce this behaviour. In this paper, we present a technique that develops passive test oracles for components using their APIs. This simple technique can be applied to any software component that is accessed through an API. In an initial experiment, we found that test oracles developed this way were more effective at finding faults with a relatively small number of test cases than test oracles developed from a formal specification and developed as a parallel implementation.

1. Introduction

Testing is an essential activity to assess the behaviour and quality of a software component. The state of the art in software testing during the past 30 years has developed numerous, often overlapping, testing methods and practices: functional testing, statistical testing, white-box testing, black-box testing, unit testing, system testing and many others. These testing approaches include both logic-driven and data-driven test case generation. Results of execution of these test cases must be evaluated to determine the correctness of the behaviour of the software component. The test result evaluation is accomplished by using a test oracle [7]. A test oracle determines whether a test case passes or fails.

Test result evaluation using a test oracle is widely acknowledged in the software testing literature as a critical aspect of the testing process. Several methods for developing test oracles, such as those using specifications [9, 13, 17], documentation [12], and parallel implementations [1], have been reported. Unfortunately development and use of such resources (specifications, documentation and parallel implementations) may require considerable effort. They can be costly to write and maintain. A limitation of using a resource to derive a test oracle is that the test oracle is only as good as the resource from which it was derived. Another drawback of some of these methods is limited applicability because documents such as formal specifications are rarely used in practice.

In addition to the above limitations, many of the test oracle development methods assume that they are developing test oracles for in-house software components where access to the internal state, specification and documentation is available. As a result, these methods cannot be applied when access to the internal state and detailed documentation is not available, for example, in the case of COTS (commercial off the shelf) components.

Hoffman and Strooper [5] define several types of test oracles including active and passive oracles. An active oracle mimics the behaviour of the software component under test. A passive oracle checks the behaviour of the component, but does not reproduce it. In this paper, we present a technique to develop a passive test oracle for a software component that uses the component's API (application programmer interface) for its behaviour-checking. The practice of using a component's API for testing is often applied on an ad-hoc basis in industry. Clearly the amount of checking that can be done with such an oracle depends on how observable the state of the component is through its public interface. Another potential danger in using the component's own API as a test oracle is that this may mask errors: the component behaves incorrectly, but the part of the component's API that is used as an oracle behaves incorrectly in exactly the same way, thus masking the error.

In this paper, we show that the approach presented has the following benefits.

Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)
0-7695-2465-6/05 $20.00 © 2005 IEEE

- Test oracle development is typically straightforward.
- Test oracles can be surprisingly effective at finding faults with a relatively small number of test cases.
- This approach does not require special documentation or separate tool support, and will work in most programming languages and component technologies.

This paper is organised as follows. Section 2 introduces the technique for passive test oracle development using the component's API. Section 3 demonstrates applying the test oracle technique on two existing software components, SymbolTable and Forest. Section 4 discusses the initial experimentation comparing the test oracles using the APIs and test oracles developed from a formal specification and as a parallel implementation. Section 5 summarises related work. Section 6 presents our conclusions and future work.

2. Technique Overview

The basic idea of the wrapper approach [2, 9, 10] is simple: place the component under test in a wrapper or decorator that takes responsibility for performing behaviour checks. Such a wrapper provides exactly the same syntactic interface as the component and assigns the work of the actual function execution to the component held inside. Figure 1 illustrates this simple idea.

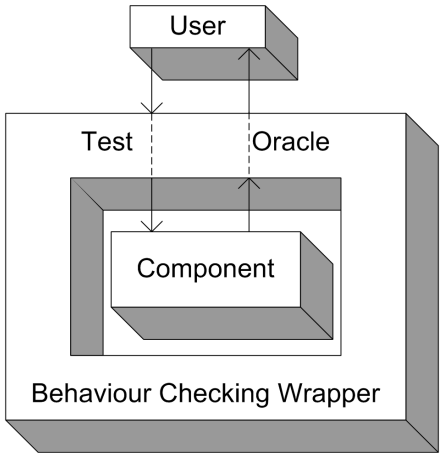

Figure 1: Wrapper approach for test oracle

The behaviour-checking wrappers are augmented versions of the publicly visible member functions of the component under test. The publicly visible interface features of the component are used to develop the interface of the wrapper component. In this way, the two components have the same externally visible features: an unwrapped original component that is the component under test for actual function execution and the (behaviour-checking) wrapper component that is a test oracle for output evaluation.

The wrapper component consists of the test oracle and wrapper functions for each of the public member functions defined in the component implementation. Each wrapper function calls the corresponding member function and then a local behaviour-checking function to check the behaviour of the implementation.

A test oracle using a component's API is one in which the component's interface is used for behaviour-checking. In this case, the test oracle checks the behaviour of the component by calling other member functions of the component using its API. While the wrapper approach supports the use of different types of behaviour-checking functions (such as those based on formal specifications and parallel implementations), we explore test oracles using the component's API in this paper.

3. Examples

The API wrapper test oracle development technique discussed above is applied to develop test oracles for the SymbolTable and Forest components of the PGMGEN testing tool [6]. PGMGEN stores exception names as symbols in SymbolTable, and then uses the list of exception names to generate exception handler code in a test driver. The table stores pairs of symbols (strings) and identifiers (integers). Symbols and identifiers must be unique. The Forest component is used to build a forest of abstract syntax trees of the input script file in PGMGEN. The Forest class is more complicated than the SymbolTable class and has operations to add new trees to the forest, to add a sub-tree as a child of another tree, and to traverse a tree. Each node has a value (the token in the input file), a type (the type of the token), and the line number on which the token occurs. Table 1 shows the source (without comments) lines of code (LOC) and number of methods for each component.

Table 1: Details of each component

Component	LOC	Number of methods
SymbolTable	128	7
Forest	234	10

The API of `SymbolTable` is shown in Figure 2. The constant `MAX_SYMBOLS` indicates that a maximum of 50 symbols are allowed in the table and

MAX_SYM_LENGTH indicates that the maximum length of a symbol is 20. The insert method adds a new symbol sym and assigns an identifier to it. The method size returns the number of pairs in the table. The existId method returns whether identifier id is in the table. Similarly existSym returns whether symbol sym occurs in the table. The method del deletes identifier id and its corresponding symbol from the table. The getSym and getId methods return the symbol and identifier for a given identifier and symbol respectively. The Java exception handling mechanism is used to signal exceptions in the implementation. The insert method throws MaxLengthExc if sym has more than MAX_SYM_LEN characters, FullExc if the table has MAX_SYMBOLS symbols in it and ExistSymExc if sym already exists in the table. The methods del and getSym throw NotExistIdExc if there is no identifier id in the table. The getId method throws NotExistSymExc if sym is not in the table.

```
public class SymbolTable {
    static final int MAX_SYMBOLS = 50;
    static final int MAX_SYM_LENGTH = 20;
    public SymbolTable();
    public void insert(String sym) throws
       MaxLengthExc, FullExc,ExistSymExc;
    public int size();
    public boolean existId(int id);
    public boolean existSym(String sym);
    public void del(int id) throws
       NotExistIdExc;
    public String getSym(int id) throws
       NotExistIdExc;
    public int getId(String sym) throws
       NotExistSymExc;
}
```

Figure 2: API for SymbolTable

The wrapper component, SymbolOracle, inherits from the implementation and checks the actual behaviour with the expected behaviour. Figure 3 shows the API of the wrapper component, SymbolOracle. The SymbolOracle contains the wrapper methods that have the same signatures as in SymbolTable, except that the oracle methods do not signal any exceptions. While the method can easily be extended to deal with exceptions, the oracles described in this paper were developed in the context of research on statistical testing [16]. In this work, components are tested according to the expected use of the component in an application, and as such we do not expect that any calls should signal an exception. Hence the oracle wrapper methods catch and print any exceptions that are signalled.

The constructor of the SymbolOracle calls the inherited component constructor and then checks its behaviour by calling the size method. The other wrapper methods perform similar checking. As an example, the implementation of the wrapper method for insert and its behaviour checker checkInsert are shown in Figure 4.

```
public class SymbolOracle extends
SymbolTable {
    // Wrapper methods
    public SymbolOracle();//constructor
    public void insert(String sym);
    public int size();
    public boolean existId(int id);
    public boolean existSym(String sym);
    public void del(int id);
    public String getSym(int id);
    public int getId(String sym);
}
```

Figure 3: API for SymbolOracle

```
public void insert(String sym) {
    int before = super.size();
    try {
         super.insert(sym);
    }
    catch (Exception e) {
        System.out.println("Unexpected
        exception in insert "+e);
    }
    int after = super.size();
    checkInsert(sym, before, after);
}

void checkInsert(String sym, int before,
int after) {
    if ((super.existSym(sym)) &&
       ((before+1) == after));
    else
         System.out.println("*** Insert()
         error ***");
}
```

Figure 4: Implementation of wrapper method insert and behaviour checker checkInsert

The method insert of the SymbolOracle calls the size method to get the size of the SymbolTable before inserting a sym. The Java exception handling mechanism is used to catch any exceptions that get thrown when the member methods are called. The wrapper method calls the inherited insert method in a try-catch block that outputs any exception that was signalled. Then the wrapper method

calls the `size` method again to get the size of the `SymbolTable` after `insert`. The input `sym` and the size before and after inserting `sym` are passed to the method behaviour checker, `checkInsert`, which checks that the input `sym` exists in the `SymbolTable` by calling the `existSym` method and that the size was incremented correctly.

The `size` and `existSym` methods are used for the behaviour-checking of the `insert` method. If a test case of `insert` fails during execution, the fault could be in the `insert`, `size` or `existSym` method. Thus, a behaviour-checking method provides double-sided behaviour-checking, to both the member method and its behaviour checker methods. The test oracle checks the behaviour of more than one method for each test case during execution. It means that the test oracle can detect more faults with a small number of test calls. However, if the methods `insert`, `size` and `existSym` are consistently incorrect then the test oracle may mask errors.

Similarly, the `getSym` and `existSym` methods are used for the behaviour-checking of the `existId` method; `getId` and `existId` for the checking of `existSym`; `size` and `existSym` for the checking of `del`; `getId` for the checking of `getSym`; and `getSym` for the checking of `getId`.

In this example, a behaviour checker method for the `size` method was not provided, as the internal state for the size method is not observable through the API of the component. Instead, the behaviour of the size method is checked when calls are made to the `insert` and `del` methods.

Despite being a more complicated component than SymbolTable, test oracle development for the Forest component using its interface is as easy as for the SymbolTable. The average size of the checking methods is 12 LOC.

4. Experiments

To compare our test oracle technique with other test oracle techniques, we implemented two additional test oracles using the wrapper approach for SymbolTable and Forest.

Following the approach in [9], we developed a passive test oracle from an Object-Z specification. In this case, the test oracle also contains an abstraction function to relate the concrete implementation state to the abstract specification state and an invariant checker to check the invariant of the component.

We also developed an active test oracle in which the state of a parallel implementation is used to generate the expected behaviour of the component [1].

Table 2 shows the source LOC of these test oracles. In both cases, the API wrapper test oracle is smaller (and simpler) than the other two test oracles. The other two test oracles use a common, named exception message to check the exception-behaviour of the component. This means that these two oracles to slightly more checking than the API oracle, but this additional checking resulted in only 12 additional lines in `SymbolTable` and 28 additional lines in `Forest`.

Table 2: LOC of each test oracle

Component	LOC of Test Oracle		
	Using specification	Active	Using API
SymbolTable	275	251	148
Forest	387	366	269

To compare the fault-detection ability of these test oracles, we use the MuJava tool [8] for fault-seeding and the STSC tool [16] for test case generation, test case execution and test output evaluation.

Fault-seeding tools such as MuJava measure the error-detection power of test cases by introducing simple faults, called mutants, into a component under test to create a set of faulty versions. These mutants are created from the original program by applying mutation operators, which describe syntactic changes in the program. Each mutant is executed with a set of test cases. When a mutant produces different output from the original software component on a test case, that mutant is said to be "killed" by that test case. Killed mutants are not executed against subsequent test cases. Some mutants cannot be killed because they are functionally equivalent to the original component. These are called equivalent mutants. The fault-detection ability of a set of test cases can then be "measured" by determining how many of the non-equivalent mutants were killed. Of course, the problem of determining which mutants are equivalent can be a difficult one.

In our experiments, we used the MuJava tool to automatically generate mutants for the `SymbolTable` and `Forest` components and we tried to find the equivalent mutants by hand (after discounting any mutants that were killed during the testing). The MuJava tool generated 188 and 242 non-equivalent mutants for `SymbolTable` and `Forest`. Note that each mutant represents exactly one fault.

The STSC tool is a statistical testing tool that generates statistically representative test cases from a model of expected operational use of the component. The STSC tool also supports a wide range of test oracles using the wrapper approach presented in this

paper for output evaluation. To evaluate the test oracles, we generated 10 different test sequences ranging from 25 to 5000 test cases for each component using a hypothetical operational use of the component.

The graphs in Figure 5 and 6 show the percentage of faults detected by the test cases using each test oracle for the SymbolTable and Forest components.

The results for SymbolTable in Figure 5 show that the passive oracle using the component's API detects more faults than the other two test oracles in the first six test sequences (up to 1000 test cases) because it checks behaviour of more than one method in each test case. Both the passive oracles detect the same percentage of faults in the seventh, eighth and ninth (2000, 3000 and 4000 test cases) test sequences. In the last, tenth, test sequence, the passive oracle using the component's API and the active oracle detect the same percentage of faults, but the passive oracle using Object-Z detects one more fault and kills all non-equivalent mutants.

Figure 6 shows that for the Forest component the passive oracle using the component's API is more effective at finding faults than the other test oracles. The passive oracle using Object-Z detects fewer faults because of a partial implementation of the abstraction function. With a full implementation of the abstraction function, we expect that the passive oracle using Object-Z would perform as effectively as the active oracle.

Further experimentation on the fault-detection ability of the test oracles is currently being carried out with different types of test cases generated using expected operational use and actual use of the components.

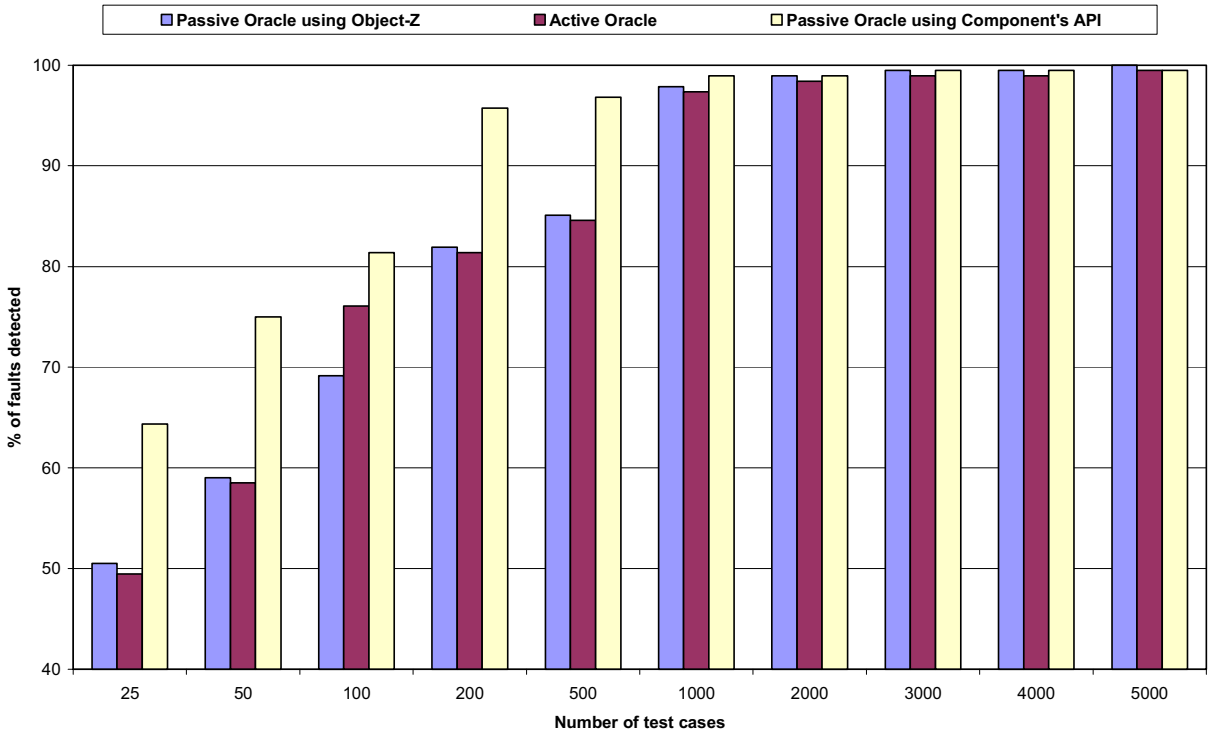

Figure 5: Fault-detection ability of the test oracles for SymbolTable

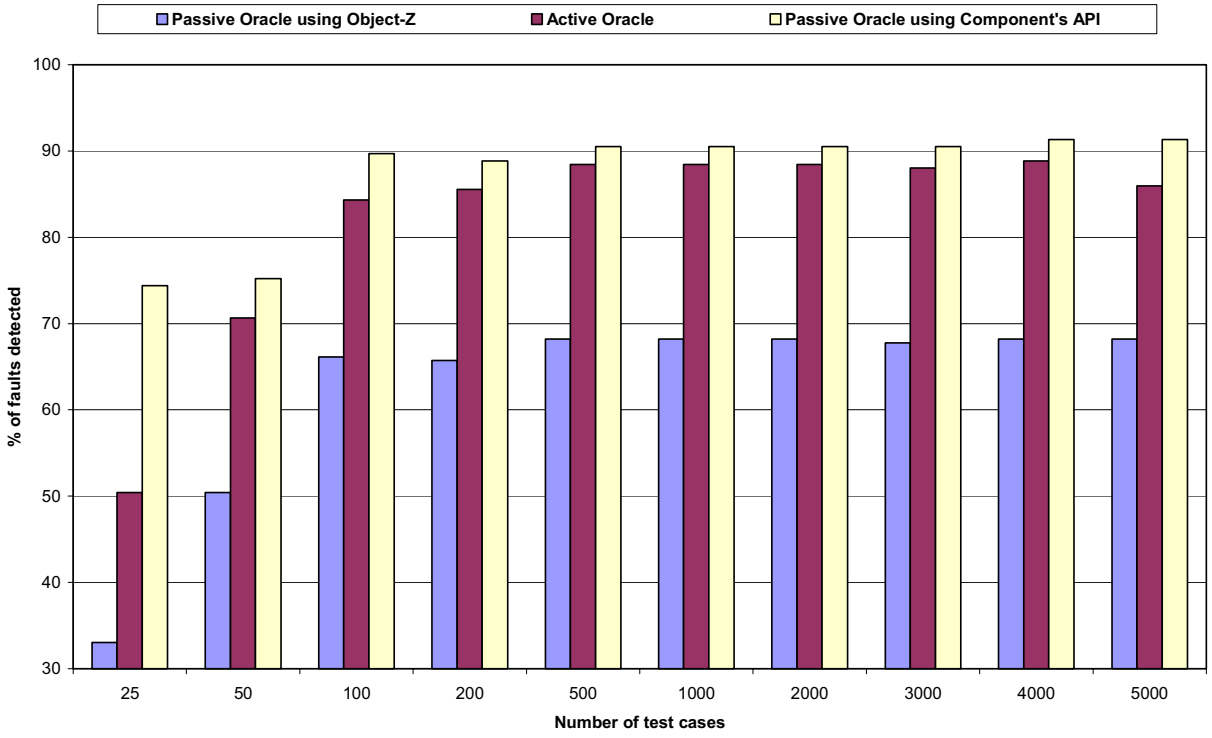

Figure 6: Fault-detection ability of the test oracles for Forest

5. Related Work

A wrapper is a component that is used to control access to a second component. The wrapper literally wraps around the second component, allowing enforcement of a higher degree of checking and security than the component can enforce on its own [10]. Many researchers have used wrappers to add assertion (pre-conditions, post-conditions and invariants) checking [2, 3], which is used to detect contract violations based on the design-by-contract principles [10]. Assertions have also been used for security (encryption, authentication, access control, intrusion detection) checking [4].

The papers most related to our work are those of Miller et al. [11] and McDonald et al. [9]. Both of these papers present methods for generating passive test oracles using the wrapper approach presented in this paper. Our approach differs from their approach because we are using a component's API for behaviour-checking instead of a formal specification.

6. Conclusions

An approach combining passive oracles implemented as a wrapper with checking functions based on the API of a software component has been presented in this paper. The technique has been applied to develop test oracles for the SymbolTable and Forest components, and was easy to implement and produced good results compared to other test oracles derived from formal specifications and parallel implementations. The technique can be applied to any type of software component in most programming languages and component technologies provided the component is accessed through an API.

This work contributes to a larger project on testing that aims to develop a framework and tool support for the statistical testing of software components [15, 16], including a method for operational profile development [14].

To test the scalability of the approach, we have started to apply it to an industrial case study using a component from an e-Healthcare system.

References

[1] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Reading, Massachusetts: Addison-Wesley, 2000.

[2] S. H. Edwards, M. Sitaraman, B. W. Weide, and J. Hollingsworth, "Contract-checking

wrappers for C++ classes," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 794-810, 2004.

[3] R. B. Findler, M. Latendresse, and M. Felleisen, "Behavioral contracts and behavioral subtyping," In Proceedings of 8th European Software Engineering Conference, pp. 229-236, 2001.

[4] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," In Proceedings of Foundations of Intrusion Tolerant Systems, pp. 399-413, 2003.

[5] D. M. Hoffman and P. A. Strooper, "Automated module testing in Prolog," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 934 -943, 1991.

[6] D. M. Hoffman and P. A. Strooper, *Software Design, Automated Testing, and Maintenance A Practical Approach*: International Thomson Computer Press, 1995.

[7] W. E. Howden and P. Eichhorst, "Proving properties of programs from program traces," in *Tutorial: software testing & validation techniques*, E. Miller and W. E. Howden, Eds., 2nd ed: IEEE Computer Society Press, New York, 1981, pp. 46-56.

[8] Y.-S. Ma, J. A. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97-133, 2005.

[9] J. McDonald and P. A. Strooper, "Translating Object-Z specifications to passive test oracles," In Proceedings of Second International Conference on Formal Engineering Methods, Brisbane, Australia, pp. 165 -174, 1998.

[10] B. Meyer, *Object-oriented software construction*, 2nd ed: Prentice Hall, 1997.

[11] T. Miller and P. A. Strooper, "Supporting the software testing process through specification animation," In Proceedings of First International Conference on Software Engineering and Formal Methods, pp. 14-23, 2003.

[12] D. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161 -173, 1998.

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based test oracles for reactive systems," In Proceedings of International Conference on Software Engineering, Melbourne, Australia, pp. 105-118, 1992.

[14] R. Y. Shukla, D. A. Carrington, and P. A. Strooper, "Systematic operational profile development for software components," In Proceedings of 11th Asia-Pacific Software Engineering Conference (APSEC), Busan, Korea, pp. 528-537, 2004.

[15] R. Y. Shukla, P. A. Strooper, and D. A. Carrington, "A framework for reliability assessment of software components," In Proceedings of 7th International Symposium on Component-based Software Engineering (CBSE), Edinburgh, UK, Lecture Notes in Computer Science, vol. 3054, Springer, Berlin, pp. 272-279, 2004.

[16] R. Y. Shukla, P. A. Strooper, and D. A. Carrington, "Tool support for statistical testing of software components," 12th Asia-Pacific Software Engineering Conference (accepted), 2005.

[17] H. Zhu, "A note on test oracles and semantics of algebraic specifications," In Proceedings of Third International Conference On Quality Software (QSIC), Dallas, Texas, pp. 91-98, 2003.