Tool Support for Statistical Testing of Software Components

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

Abstract

We describe the "STSC" prototype tool that supports the statistical testing of software components. The tool supports a wide range of operational profiles and test oracles for test case generation and output evaluation. The tool also generates appropriate values for different types of input parameters of operations. STSC automatically generates a test driver from an operational profile. This test driver invokes a test oracle that is implemented as a behaviour-checking version of the implementation. To evaluate the flexibility and usability of the tool, it has been applied to several case studies using different types of operational profiles and test oracles.

1. Introduction

Software now controls banking systems, telecommunication systems, transport systems, factories, automobiles and even household gadgets. Society has developed an extra-ordinary dependence on software. Hence, software failure is regarded as one of the most important concerns of software in everyday lives. Extensive and efficient testing of these software systems and their components is considered an obvious prerequisite for high quality software. For the development of high-quality software with certified reliability, Cleanroom software engineering uses software testing based on statistical principles [12]. Statistical usage testing involves testing software the way the users use it and focuses on external behaviour, not the internals of the software [14].

Statistical testing of a software component from a user's point of view depends largely on the manner in which the component is used. Characterisation of the population of expected use is referred to as an operational profile. An operational profile is a set of input events and their associated probabilities of occurrence expected in actual operation. The test cases that are executed during a statistical test are a sample from the operational profile. The result of testing obtained in this way depends upon accuracy of the operational profile.

Test output evaluation is a difficult and important problem for statistical testing. An expected result is needed for each test case to check the test output. The mechanism used to check these expected results is called a test oracle. A test oracle is an essential part of statistical testing, because a wide range and large number of test cases are required and the behaviour must be checked for every test case.

In an earlier paper, we incorporated test oracles with statistical testing and proposed a conceptual framework for the statistical testing of software components [18]. In this paper, we describe the Statistical Testing for Software Components (STSC) prototype tool that supports the framework for test case generation, test case execution and output evaluation. STSC supports a wide range of operational profile approaches for test case generation and a variety of test oracles for output evaluation. STSC also supports generation of appropriate values for different types of input parameters of operations. The tool automatically generates a test driver from an operational profile. This test driver invokes a test oracle that is implemented as a behaviour-checking version of the component under test, which calls the implementation and then checks the results produced by the component. To evaluate the flexibility and usability of the STSC, we apply the tool to a simple Stack and two existing components, SymbolTable and Forest (of abstract syntax trees), that are used in the implementation of the PGMGEN tool [8].

The paper is organised as follows. Section 2 discusses the issues of statistical testing addressed by the tool. Section 3 introduces the STSC tool. Section 4 discusses experience with using STSC on the Stack, SymbolTable and Forest components. Section 5 summarises related work. Section 6 presents our conclusions and future work.

2. Issues

The simplest form of operational profile is a uniform distribution in which the probability of occurrence for each of the operations is equal. Such a uniform distribution is easy to model and implement in a test case generator. However, non-uniform distributions are typically encountered in real applications and must be supported by a tool for statistical testing. A wide variety of modelling notations, such as Markov chains [19], state machines [11], UML [15], and probabilistic statecharts [16] have been reported for statistical testing, and a number of tools based on specific usage models have been reported. For example, MaTeLo [9] is based on Markov chain models and Riebisch et al. [15] describe a statistical test case generator based on UML. We designed STSC to be general enough to support any of the above operational profiles.

One aspect in which operational profiles differ is in the determination of the number and termination of test sequences. For example, some operational profiles explicitly define when a test sequence is complete by including final state(s) in the model of the operational profile. In other cases, the decision as to when to terminate one test sequence and start another is not part of the operational profile and left as a decision for the test case generator. We designed STSC so that both approaches to splitting up test cases into test sequences are supported.

Most of the research on operational profiles has focused on operations and little is said about operation parameters. Random values for input parameters are a common practice in statistical testing. The entire domain of the input parameter is considered and the test generator randomly selects values from this domain. For generation of a single independent input value, random generators may provide an adequate solution. Each randomly generated value is unrelated to the next randomly generated value. However, related input data sequences cannot be generated this way. The sequence of test cases generated from a test generator would be meaningless when operation parameters values are not consistent with parameter values from the expected usage.

For software components, we have found that assigning appropriate values for input parameters can be quite complicated, because they depend not only on the type of the parameter, but there are often also constraints on individual parameters and intricate relationships between different input parameters (to the same or even different operation calls) and between output parameters of calls and input parameters of subsequent calls. In an earlier paper, we described a method for deriving those constraints and relationships to generate appropriate input parameters values as part of the operational profile [16]. In the STSC tool, we provide a facility to support the flexible generation of input parameters so that constraints on and relationships between parameters are satisfied.

A test oracle is needed for each test case to evaluate the test behaviour. A number of statistical testing researchers assume that a test oracle is available for output evaluation [11, 13, 20]. Several test oracle methods, for example using a formal specification [10] or a parallel implementation [1], are reported in the software testing literature. In addition, we want the framework and tool support to be applicable for the statistical testing of third-party components, such as commercial off the shelf (COTS) components. For such components, the test oracle will typically not have access to the internals of the implementation. The STSC tool was designed to support all these types of test oracles.

3. STSC Tool

Figure 1 shows an overview of our framework [18] for statistical testing. The rectangles represent processes, the ovals represent outputs, the folded corner documents represent inputs and the cubes represent software components. Test case generation requires the definition of an operational profile and a number of testing parameters.

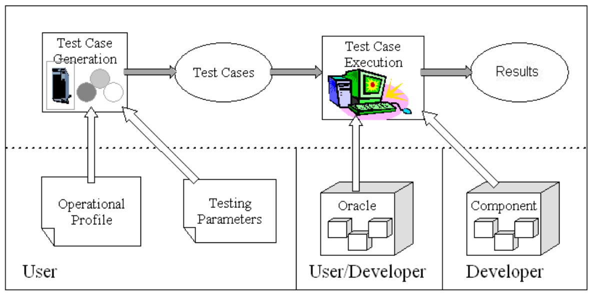

Figure 1: Framework for statistical testing

In this section we discuss the STSC tool developed for statistical testing of software components. Matching the framework in Figure 1, the STSC tool supports: (1) test case generation; and (2) test case execution. Test case generation is supported by the TCG tool, which samples from the operational profile to generate test cases, executes these test cases, and evaluates the test output. The TCG uses the operational profile and testing parameters, and generates a test driver. When this test driver is executed, a test oracle is needed to check the results produced by the component under test.

The stakeholders are the software component user and the software component developer. The tool requires the user to specify an operational profile and the testing parameters. The component developer supplies the component. The test oracle can be written by the developer or by the user. The latter will have to be the case when the developer does not supply an oracle, as is typically the case for COTS components.

3.1. Test Case Generation

To allow the generation of test cases from a wide variety of operational profiles and to deal with the issues discussed in the previous section, the TCG is implemented as a *software framework*, as defined by Gamma et al. [4], which is a reusable software component that requires other software components as input to perform its function. In this case, the TCG relies on a software component that must implement OPinterface (Operational Profile interface). This OP component would typically be implemented by the software component user who is also the person interested in testing the component. As such, this person would typically be a programmer. The operational profile is specified by the combination of the OP component and a distribution table that defines one or more probability distributions for the operations in the component under test. In addition, the TCG takes a number of testing parameters as input.

With this approach, the distribution table and the implementation of the OP component will be simple for straightforward operational profiles. For example, for an unconditional operational profile, for which there is only a single, unconditional probability distribution for all the operations of the component, the distribution table would only have to define this single probability distribution. For a conditional operational profile that contains a number of usage conditions, each of which defines a distinct way of using the component with an associated probability distribution, the distribution table would define the probability distribution for each usage condition. Similarly, for components for which it is easy to generate the input parameters of operations and for which there are no interdependencies between the parameters of different calls, the implementation of the input and output parameter methods (as detailed below) of the OP component would be trivial.

Using the testing parameters and the information in the distribution table, and by making calls to the OP component implemented by the software component user, the TCG generates a Java test driver that contains calls to the component under test according to the operational profile. The TCG also produces a significance level [7] of the generated test cases, which can be used to determine the statistical significance of the test cases.

3.1.1. Testing Parameters. The command-line parameters to the TCG are:

- `calls`: number of calls per sequence to be generated in the output test driver;
- `sequences`: number of sequences to be generated in the output test driver;
- `seed`: a seed for random number generation;
- `distribution table`: the name of the input distribution table file;
- `OP component`: the name of component that implements OPinterface;
- `oracle component`: the name of the test oracle component;
- `frequency file`: output file name in which the frequency of actual operations generated in the output test driver and the significance level of the test cases will be stored; and
- `test driver`: output file in which the Java test driver will be stored.

The user can provide both `calls` and `sequences` as parameters to specify the number of calls per sequence to be generated in the test driver. The sequences and seed parameters are optional. If the tool is run twice with the same `seed` and the same testing parameters, it will generate identical output frequency files and test drivers. This is so that we can generate identical test drivers (e.g. for debugging).

3.1.2. Distribution Table. The TCG generates test calls according to the probability distribution described in the distribution table. The distribution table describes the number of usage conditions, the number of operations, the name of the operations and a probability distribution for each usage condition of the operational profile.

3.1.3. OP Component. The Java interface `OPinterface`, shown in Figure 2, defines the methods that must be implemented in the OP component for the TCG to function properly. We discuss each method in turn.

The `checkTestingPara` has two input parameters from the testing parameters: `calls` and `sequences`. The method checks the `calls` and `sequences` with the actual requirement of the calls and sequences for that particular operational profile and returns true if the specified parameters are valid.

The checking process may contain limit checks on the parameters.

```
public interface OPinterface {
    // checks testing parameters
    boolean checkTestingPara(int calls,
      int sequences);
    // Transition function
    int transition(int uc, int on);
    // Input function
    String inputpara(int uc, int on);
    // Output function
    String outputpara(int uc, int on);
}
```

Figure 2: Interface for OP component

The other three methods in `OPinterface` have two input parameters: the current usage condition `uc` in the distribution table and the current operation number `on` (each operation in the component under test has a unique number associated with it) that is generated in the test driver.

Based on the values of `uc` and `on`, the `transition` method returns the next usage condition (represented as an integer) to use after the current call. It also informs the TCG when a new test sequence must be started by returning the special value of -1. For the special case of an unconditional probability distribution, this method always returns the value 1, unless a new test sequence must be started. Determining the next usage condition is much harder for complicated operational profiles, e.g., those modelled using statecharts [16].

The `inputpara` and `outputpara` methods support the generation of appropriate values for input parameters and allow constraints between different parameters to be observed. The `inputpara` method must return a string that represents the input parameters to be used in the generated test driver for the current operation. If there is more than one input parameter for the current operation, the method must return them as a single string, using commas to separate the parameters. A special `utility` library component that is part of TCG can be used to easily generate random values of various types (integer, long, float, double, Boolean, character and String) for input parameters. If there are no constraints on input and output parameters and the parameters can be easily generated as random values, then the `inputpara` method would simply contain calls to these `utility` methods based on the current operation number and the `outputpara` method is not needed.

When there are constraints on and relationships between operation parameters, the situation is much more complicated, because the actual parameter values that are returned by operations of the component under test are only known when the generated test driver is executed, and not when the TCG runs to generate the test driver. To deal with this situation, the `outputpara` method can be used to define variables in the generated test driver to capture the return values of calls so that they can later be used as input parameters to other calls. To do this, the `outputpara` method returns a string, which is prepended to the current operation call in the generated test driver. Similarly, if there are relationships between different input parameters of the same or even different calls, this can be implemented through the `inputpara` method.

Given the testing parameters, the distribution table, and the OP Component, the TCG framework executes the pseudo-code shown in Figure 3.

```
if (checkTestingPara(calls, sequences)) {
    tnc = 0; // total number of calls
    ns = -1; // number of sequences
    nc = 0;  // number of calls per sequence
    uc = -1; // usage condition
    while not_finished_generating_calls {
       if ((nc = calls) OR (uc = -1)) { // start new test sequence
           uc = 1;
           instantiate new component under test in test driver;
           ns++;              nc = 0;
       }
       on = randomly generate operation number from distribution table based on uc;
       use on,inputpara(uc,on), and outputpara(uc,on) to generate call in test driver;
       nc++;         tnc++;
       uc = transition(uc, on);
    }
} else
    report error;
}
```

Figure 3: Pseudo-code executed by TCG framework

The `checkTestingPara` is called once at the beginning of the test case generation with the testing parameters `calls` and `sequences` passed as arguments. A special value of -1 is used for sequences, if this parameter was not specified (the number of calls must always be specified). The method should return true if and only if the specified test parameters are valid. For example, for operational profiles that have explicit final states that can be used to determine when a test sequence is complete, only the number of calls can be specified and the TCG will generate calls based on the operational profile until this total number of calls has been reached (starting a new test sequence when a final state in the operational model is reached). On the other hand, if no such final states exist, the user could specify both the number of test sequences and the number of calls, which in this case can be interpreted as the number of calls per test sequence. Other combinations and interpretations of the test parameters `calls` and `sequences` are possible, and can be used in combination with the definition of the transition method to control the number of test sequences and calls generated. If the `checkTestingPara` method returns false, the TCG terminates and does not generate any test cases.

Note that with the above approach, once the operational profile has been defined in terms of the distribution table and the OP Component, a large number of test drivers with different numbers of test sequences and/or calls can easily be generated from this operational profile by simply varying the testing parameters.

3.2. Test Case Execution

Test outputs of the component under test must be evaluated during testing. This is done through a test oracle that provides a behaviour-checking implementation of the component, which can be implemented using inheritance or delegation [18]. The test oracle is a wrapper or decorator [4] around the component under test. The test oracle presents the same user interface as the component under test and is used in place of the component during test execution. During test case execution, the generated test driver is executed and the test output is evaluated. The test driver invokes the supplied test oracle that invokes the component under test and then checks its behaviour. The result of the test case execution and output evaluation allows us to confirm or deny that the component behaves correctly.

4. Case Studies

The tool supports a wide range of operational profiles for test case generation and a variety of test oracles for output evaluation. To check the practical viability and flexibility of STSC, we applied the tool to the Stack, SymbolTable and Forest (of abstract syntax trees) components using different types of operational profiles and test oracles.

4.1. Subject Components

Table 1 shows the source (without comments) lines of code (LOC), number of operations and number of input parameters of operations for the three components.

Table 1: Details of each component

Component	LOC	Number of operations	Number of input parameters
Stack	35	3	1
SymbolTable	128	7	7
Forest	234	10	14

We use the Stack from [20] as an initial case study to check the working of the tool.

SymbolTable and Forest are more realistic components from an existing system, the PGMGEN testing tool [8]. 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 SymbolTable stores pairs of symbols (strings) and identifiers (integers).

The Forest component is more complex than SymbolTable and is used to build and access a forest of abstract syntax trees of the input script file in PGMGEN. Generation of appropriate values for input parameters of the operations is more complex for this component because of intricate relationships between parameters. In particular, random values for input parameters will result in meaningless calls that only generate exceptions.

4.2. Operational Profiles

We use the hypothetical operational profile presented in [20] for the Stack component. The LOC of our OP component is 59 while the LOC of the Woit's userfiles [20] in her prototype tool is 93. Note that Woit's driver only works for the Stack component, and would need to be updated to generate statistical tests for other components.

We implemented four different operational profiles as described below for the SymbolTable and Forest components. The first two operational profiles are hypothetically generated and the last two are developed using actual usage data from the components when used as part of PGMGEN. The first two operational profiles use randomly generated values for input parameters of operations and the remaining operational profiles generate/assign appropriate values for input parameters of operations from the constraints and relationships between them as derived from the component's use in PGMGEN.

An unconditional uniformly distributed operational profile is one in which the probability of occurrence of each operation is equal. Since the profile is unconditional, there is only one usage condition. The probability vector contains equal probability for each operation.

A conditional uniformly distributed operational profile is one in which the probability of occurrence of each enabled operation for each usage condition is equal. The difference with the unconditional uniformly distributed operational profile is that for different usage conditions, different subsets of the set of operations of the component are "enabled" and the other operations that are not enabled for a particular usage condition cannot be selected for that usage condition. In the case of SymbolTable, we assume that there are three usage conditions: create symbols, list the symbols and find the identifiers of the symbols.

The third operational profile is described using a Markov model in which the probabilities of the operations are based on the last call issued. This operational profile is derived by applying the bottom-up approach presented in [16] using both actual usage data and intended usage assumptions.

The fourth operational profile is described using probabilistic statecharts in which the probability of an input event depends on any or all prior input events. This operational profile is derived by applying the top-down approach presented in [16] using both actual usage data and intended usage assumptions. The probability of occurrence of each operation for each usage state is calculated from the usage data.

Table 2 shows the number of usage conditions and source LOC of OP components for these operational profiles for each subject component.

As indicated earlier, because of dependencies between parameters, the test drivers generated from the first two operational profiles for the Forest component do not generate many valid calls, as most calls signal an exception. As a result, these drivers are not representative of the actual use of this component (which shows that more sophisticated operational profiles are necessary for the statistical testing of this component).

Table 2: Details of each operational profile

Component	Uniformly distributed Operational Profile			
	Unconditional		Conditional	
	Usage conditions	LOC	Usage conditions	LOC
SymbolTable	1	46	3	82
Forest	1	59	2	70

Component	Operational Profile described using			
	Markov model		Probabilistic statecharts	
	Usage conditions	LOC	Usage conditions	LOC
Stack	---	---	3	59
SymbolTable	7	81	15	133
Forest	12	215	16	273

4.3. Test Oracles

Following the approach in [10], we developed a passive test oracle from the Object-Z [3] specification for the Stack component, where the abstraction function relates the concrete implementation state to an abstract state and predicates from the Object-Z specification that is modelled using classes from the Java JDK. The approach can only be applied to in-house components, in which the test oracle can access the internal state and Object-Z specification.

In addition to the above test oracle, we implemented two additional test oracles for SymbolTable and Forest. These additional test oracles do not rely on knowledge of the implementation, and as such are more representative of the types of oracles that are needed for third-party (e.g. COTS) components.

A test oracle using a component's API (application programmer interface) is one in which the component's interface is used to check the behaviour of the component [17]. Clearly the amount of checking that can be done with such an oracle depends on how observable the internal state of the component is through its public interface.

An active test oracle is one in which the state of a parallel implementation is used to generate the expected behaviour of the implementation [1]. Such an approach to test oracle development involves implementing a second version of the component. Clearly this can be prohibitively expensive but since the oracle does not need to be efficient, it may be substantially simpler than the original implementation.

Table 3 shows the source LOC of these test oracles for each subject component. The test oracles using the component's API are smaller than the other test oracles.

Table 3: LOC of each test oracle

Component	LOC of Test Oracle		
	Using specification	Using API	Active
Stack	116	---	---
SymbolTable	275	148	251
Forest	387	269	366

Initial data indicates that testing using the component's API does surprisingly well with a relatively small number of test cases [17].

5. Related Work

Model-based test generation has become an area of active research. However, relatively little research has been performed on statistical testing using models. Most of the model-based test generation focuses on obtaining test cases directly from diagrams prepared during early stages of development or for fault fixing rather than to derive usage models to guide a test case generator for statistical testing. Riebisch et al. [15] derive a Markov usage model from a UML use case model for automated generation of test cases for quality assurance during the software development process. Their transformation process derives usage models from use cases, state diagrams and usage graphs. Le Guen et al. [9] produce test cases based on usage models described using Markov models with the MaTeLo tool. The tool is developed based on the Markov model approach presented by Whittaker and Thomason [19]. However, the Markov model has limitations in describing complex behaviours. Woit [20] presents a statistical test case generator for a hypothetically generated operational profile for the earlier presented Stack component. In Woit's work, a separate test case generator is needed for each component under test. Popovic and Velikik [13] present a test case generator based on the test case generator presented by Woit [20] for their own Generic Model Environment. None of the above test generators support treatment of parameters of operations, test case execution and output evaluation.

The issue of appropriate operation parameter values has been largely ignored by the operational profile and statistical test case generation literature [11, 13, 19, 20]. Giltens et al. [5] follow Woit [20] and include a data profile, the minimum and maximum data values of the inputs to the application, in the operational profile. The problem of input parameter values is discussed in [2, 6], but no general solution is presented. Chen et al. [2] mention that it is not straightforward to generate test cases that take complex data structures as input. They suggest that input parameters can be generated by a hybrid approach using partitions of domains and random values, but do not demonstrate their approach.

6. Conclusions

We have presented the STSC prototype tool for statistical testing of software components (including COTS components). Although the framework for statistical testing has been presented previously, the tool support presented in this paper is essential to reduce the time and the potential human error in: (1) writing test drivers; and (2) manual output evaluation.

The STSC tool has been applied successfully to the Stack, SymbolTable and Forest components for test case generation, test case execution and output evaluation using different types of operational profiles and test oracles. The tool needs an operational profile and a test oracle. The tool uses the operational profile in the form of a distribution table and an implementation of various methods, and the test oracle as a wrapper around the component under test.

The operational profiles implemented for the components show that the STSC tool is flexible enough to support different types of operational profiles, including those described using Markov models and probabilistic statecharts. The test oracles implemented for the components show that the STSC tool is general enough to support a range of test oracles implemented as wrappers, including those generated from a formal specification. The test drivers with different types of operation parameter values generated from different types of constraints and relationships between them show that the STSC is generating appropriate input parameter values as per the expected usage.

We have applied four different operational profile approaches and three different test oracle techniques. However, we are interested in an accurate operational profile that represents the actual use of the component and an effective test oracle that detects all the possible faults for statistical testing. An empirical evaluation to compare the accuracy of these operational profiles and effectiveness of these test oracles is currently being carried out.

To verify the scalability of the tool, the tool is currently being applied to an industrial case study. The component has been selected from an e-Healthcare system. Initially, we plan to create an operational profile from the expected usage and a test oracle using the component's API.

References

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

[2] T. Y. Chen, G. Eddy, R. Merkel, and P. K. Wong, "Adaptive random testing through dynamic partitioning," In Proceedings of Fourth International Conference On Quality Software, pp. 79-86, 2004.

[3] R. W. Duke and G. Rose, *Formal object-oriented specification using Object-Z*: Macmillan Press Limited, 2000.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Sydney: Addison-Wesley, 1994.

[5] M. Gittens, H. Lutfiyya, and M. Bauer, "An extended operational profile model," In Proceedings of 15th International Symposium on Software Reliability Engineering, pp. 314-325, 2004.

[6] D. Hamlet, D. Mason, and D. Woit, "Theory of software reliability based on components," In Proceedings of 23rd International Conference on Software Engineering (ICSE), Toronto, Ontario, Canada, pp. 361-370, 2001.

[7] A. J. Hayter, *Probability and statistics for engineers and scientists*. Boston: PWS Co., 1996.

[8] D. M. Hoffman and P. A. Strooper, *Software design, automated testing, and maintenance a practical approach*: International Thomson Computer Press, 1995.

[9] H. Le Guen, R. Marie, and T. Thelin, "Reliability estimation for statistical usage testing using Markov chains," In Proceedings of 15th International Symposium on Software Reliability Engineering, pp. 54-65, 2004.

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

[11] J. D. McGregor, J. A. Stafford, and I.-H. Cho, "Measuring component reliability," In Proceedings of 6th ICSE Workshop on Component-based Software Engineering, 2003.

[12] H. D. Mills, "Certifying the correctness of software," In Proceedings of Twenty-Fifth Hawaii International Conference on System Sciences, Kauai, HI, USA, pp. 373-381 vol.2, 1992.

[13] M. Popovic and I. Velikic, "A generic model-based test case generator," In Proceedings of 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 221-228, 2005.

[14] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom software engineering: technology and process*. Reading, Mass: Addison-Wesley, 1999.

[15] M. Riebisch, I. Philippow, and M. Götze, "UML-based statistical test case generation," In Proceedings of International Conference NetObjectDays, Lecture Notes in Computer Science, Vol. 2591, pp. 394-411, 2003.

[16] 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, pp. 528-537, 2004.

[17] R. Y. Shukla, D. A. Carrington, and P. A. Strooper, "A passive test oracle using a component's API," 12th Asia-Pacific Software Engineering Conference (accepted), 2005.

[18] 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, pp. 272-279, 2004.

[19] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812-824, 1994.

[20] D. Woit, Operational profile specification, test case generation, and reliability estimation for modules, PhD, in *Computing and Information Science*. Canada: Queen's University, 1994.