Proceedings of the Workshop on Model-Based Testing and Object-Oriented Systems

Wolfgang Grieskamp[1] Debra Richardson[2]
Clay Williams[3]

October 2006

Technical Report
MSR-TR-2006-148

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]Microsoft Research
[2]University of California – Irvine
[3]IBM T.J. Watson Research Center

Preface

The 1st Workshop on Model-Based Testing for Object-Oriented Systems (M-TOOS) was held as part of the 2006 Object-Oriented Programming Systems, Languages, and Architectures (OOPSLA) Conference. This volume contains the final version of five papers that were accepted for the workshop, among nine which have been submitted.

Model-based testing of systems has long been a goal of the testing research community, and many paradigms have been proposed and developed. However, while model-based testing approaches are successfully used within portions of the software industry, they have yet to be adopted as mainstream practices. The M-TOOS workshop sought to explore the impact of object-orientation on model-based testing, with two primary purposes: (1) developing an understanding of the key challenges inhibiting widespread use of model-based testing approaches for testing object oriented software, and (2) determining possible ways that object-orientation (and related approaches) may be helpful in overcoming these challenges.

The workshop was structured around three main activities. First, we heard from the authors of the five accepted papers. Their contributions reflected an interesting set of topics in testing, including: the influence of APIs on testing (both from an object-oriented and multi-language point of view), testing from specifications such as scenarios, state machines, and Petri nets, and generating test data without specifications using dynamic analysis techniques.

Next, we had a panel with testing experts from both academia and industry. The topic was "How do we make model-based testing pervasive?".

Finally, the workshop concluded with a brainstorming session that sought to address the challenges that emerged from the papers and the panel.

We would like to thank our excellent program committee (listed below) as well as all of the people who submitted papers to the workshop and who attended it. Finally, we want to thank the OOPSLA organizers for their support as we planned this workshop.

Wolfgang Grieskamp, Debra Richardson, and Clay Williams
October 2006

Program Committee

- Lionel Briand, Carleton University, Canada
- Wolfgang Grieskamp, Microsoft Research, USA
- Antti Huima, Conformiq Software Ltd., Finland
- Atif Memon, University of Maryland, USA
- Henry Muccini, Universita dell'Aquila, Italy
- Alexandre Petrenko, CRIM, Canada
- Debra Richardson, University of California - Irvine, USA
- Harry Robinson, Google, USA
- Vlad Rusu, INRIA, France
- Mark Utting, University of Waikato, New Zealand
- Willem Visser, RIACS/NASA Ames Research Center, USA
- Clay Williams, IBM T.J. Watson Research Center, USA

Contents

Model-Based Testing for Object-Oriented Programming Interfaces

Ian Craggs
IBM United Kingdom
MP211 Hursley Park
Winchester, Hants. SO21 2JN
UK
icraggs@uk.ibm.com

Conor Beverland
IBM United Kingdom
MP188 Hursley Park
Winchester, Hants. SO21 2JN
UK
c.beverland@uk.ibm.com

ABSTRACT

Although model-based testing of systems has seen significant amounts of research in academia, and modest success in industry, the approach has not yet been widely adopted. We discuss the reasons for this, amongst them the cost of oracle building and the efficiency of test generation.

We describe potential solutions to these obstacles that we have been experimenting with. We outline a tool, Rule-Based Testing, which incorporates these solutions, and its use in testing Java^{TM} APIs, and present some preliminary, favourable results.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Experimentation Measurement Reliability

Keywords
Modelling, Java, Python, Test Selection, Test Generation, Model-Based Testing, Model-Driven Testing, Category-Partition Testing

1. INTRODUCTION

Model-based testing is not a specific enough term to describe what we really mean. Bach et al pronounce 'All testing is based on models'[6], which means that any testing could be labelled as model-based. When a tester creates a test suite in the traditional manner, by forming tests from selected inputs followed by the expected results, they use a mental model of the system being tested to predict those results.

That mental model is built from any materials that the tester has to hand. If they are lucky they may have some clear specifications to work from. More commonly they will extract information from a combination of sources: design documents, talking to people responsible for the design and implementation directly, customers, user documentation, and experimentation. The resulting model is never experienced directly by any other people than the creator. It is expressed in the tests that arise from it.

What we mean by 'Model-based testing' (MBT) is that the model is explicit, in the sense that it can be used by a computer to generate individual test cases and compile test suites. Some MBT tools require that the model be written using a formal method, or the UML. We specify here only that the model can serve as a test oracle, that is for any test data inputs that we choose, it can automatically predict the results to the degree of accuracy we need. This means we are free to choose the most effective oracle, regardless of the method of construction.

1.1 Java Programming Interfaces

We have used Application Programming Interfaces (APIs) written in the Java programming language as subjects for the projects described in this paper. This is primarily because many projects in our company, IBM, are implemented in Java. The language has some capabilities and conventions which make it more convenient to test with model-based methods than older languages such as C++. Reflection allows the structure of programs and APIs to be queried automatically. The standard use of the Javadoc documentation tool gives extra useful information such as the names of parameters as well as their types.

The JUnit test tool has brought about an upturn in unit testing for systems written in Java. JUnit test cases are mostly hand written sequences of inputs and expected results, the sort of test suites we are trying to move away from in adopting model-based testing. We know of several projects with a commendable focus on unit testing whose JUnit test suites comprise more code than the implementations. One such example is included in the results section 6.

Adding constraints in the manner of Design by Contract would be a logical step beyond Javadoc. This would allow for some of the unit testing to be automated by enabling the generation of test data which would be expected to violate or be acceptable to preconditions. At the functional level the constraints could help build the test oracle. Tool support would be crucial: one attempt to bring Design by Contract to Java is the Java Modelling Language (JML), which in our experience so far is hampered by the lack of a way to get started easily.

2. RULE-BASED TESTING (RBT)

Instead of using a specialised modelling language, RBT uses Python^{TM} (although this does not preclude the use of others). Python is a modern programming language, with reliable interpreters courtesy of open-source development, tested by the many people who download and use it. Python has the advantages over specialised modelling languages of being relatively well known and therefore familiar, free, and fully-featured. By many it is also viewed as easy to learn and use.

The name Rule-Based Testing was chosen because the test generator consists of a set of rules, one describing each possible operation. Each rule defines when that operation can be selected, the data choices available, and how the results of that operation are checked.

2.1 Test-Generator Generation

Information needed for test generation is extracted from the Javadoc for the API to be tested. It includes:

- Classes and interfaces and inheritance relationships between them.
- Methods, static methods and constructors.
- Parameters, their names and types.
- The types of return values.

In principle it makes no difference whether this information is retrieved from program source, documentation (Javadoc) or by reflection from compiled classes. It is purely a matter of practicality to choose one or the other.

The structural information about the API is then combined with test data selections for parameters and return values to produce a test generator. Optionally usage models may be constructed to further guide the test sequencing. All selections of the order of operations and the values of parameters are made on the basis of the externals of the API. This is akin to automating the approach of the black-box tester, and means that the modelling language need not support any form of automated structural analysis.

2.2 Test Generation

The aim of the test generation is to call all methods, use each choice specified for each parameter, and elicit all choices for return values, at least once. When several choices of test inputs carry equal weight in achieving the coverage goals, the choice is made randomly. Because we are using the API externals only in our test input selection, the first aim is easy to fulfil, the second moderately easy, and the last can be difficult - hence the use of usage models.

To predict the results of operations, an oracle is needed. We have allowed for a significant flexibility in the selection of an oracle, ranging from a specially written Python program to a another implementation (a previous release for instance). We can use another implementation *because* we do not carry out any analysis of the internal structure of the oracle. It is our proposition that in the absence of a formal model written before the test phase, this is desirable, notwithstanding the resulting relative difficulty of achieving high output coverage.

2.3 Test Execution

Tests can be selected on the fly at test generation time and the chosen action executed against both the system under test and the test oracle. Alternatively JUnit test programs which include the expected results can be generated. Either way the results, if any, are compared using the test comparator and if there is a difference, this is reported.

When we get an error, there are several options for how to go about resolving it depending on the cause:

Defect in the results predictor. Update the oracle so that the test will pass in future.

Defect in the system. Have the defect fixed.

Results were actually equivalent. Fix the comparator. An example of when this might be required would be if a method returns an array where the ordering is unspecified.

Finally, it may be decided that the results of this method need not be tested, in which case we can omit the test on future runs.

A significant help in the implementation of on the fly test execution is the existence of Jython, a Python interpreter written in Java. Jython makes using Java packages from Python programs as easy as using external Python modules. The cost of interfacing generated tests to the systems to be tested drops to near zero.

3. ADOPTION CHALLENGES

There are many challenges which any model based testing tool must meet if it is to become widely accepted. In this section we will describe those which we feel are of most importance and outline briefly how the RBT tool overcomes them.

3.1 Measurement of Success

Learning a new tool or technique actually lowers programmer productivity and product quality initially. The eventual benefit is achieved only after this learning curve is overcome, Therefore it is worth adopting new tools and techniques, but only (a) if their value is seen realistically and (b) patience is used in measuring benefits[5].

Measuring the effect of using a new tool or technique presents a significant challenge for software testing, because many organisations do not measure the effectiveness of what they do now. Often the only metric is the percentage of successful tests. If no attempt is made to measure the coverage of the test suite, then the meaning of a percentage pass rate rests entirely on the content of the tests. Ultimately that means the capability of the test designers. The accuracy of the percentage pass rate figure may often be illusory, but nevertheless many test managers rely on it heavily.

The best measure of the effectiveness of a model-based testing technique would be to compare it directly to a traditional testing approach in a side-by-side experiment. This is unlikely to be carried out in an industrial context, because of the extra cost. An alternative is to compare the quality of test suites created by the two different approaches. As standard programming interfaces become more common with the rise of Java and .Net environments for example, then the existence of standard test suites also becomes more common. A standard test suite can be used as a benchmark against which a model-based approach must compete. We intend to carry out such a comparison in the near future.

3.2 Speed

Optimally the software design process would produce artifacts which could be used directly in model-based testing. For instance, executable models of the system used for simulation and validation of the design could be used to predict expected results for tests (that is, to be test *oracles*). However this is hardly, if ever the case at present, and for any model-based testing approach to be successful, we must be able to use only existing information sources.

This means that any models required by model-based testing must be created or obtained within the timescales allowed for existing software testing methods. Models created at design time serve multiple purposes, one of the main ones being to be a communication medium for ideas. Although this is also a desirable aim for test models, the test model ultimately has another absolutely crucial purpose; the generation of tests. The pursuit of one aim may have to be sacrificed for the achievement of the other.

RBT addresses the challenge of speed by:

- Extracting as much information as possible from existing sources (Javadoc).
- Allowing many types of oracle building, so the most effective method can be chosen.
- When an oracle needs to be written, allowing Python to be used as the modelling language.

3.3 Scalability

Any test generation method should be fast enough to provide timely results, and not just on small examples, but on industrial scale systems. It is possible for test generation methods based on model-checking techniques[4] to expend hours or days on model analysis and test generation. Even then the resulting tests may be insufficient or nonexistent. For mainstream acceptance, model-based testing approaches must be designed to work on an industrial scale from the start[3]. RBT generates tests right from the start of the test generation process. There is always some useful output, even if this is a partially complete test suite.

The basic coverage measure we use in RBT scales linearly relative to the size of the API, but is less comprehensive than transition coverage of a state machine for example. Given that existing test suites and methods usually achieve low coverage by any measure, test suites generated by model-based methods do not have to have perfect coverage to be significantly better. RBT test suites have been shown to be adequate in terms of coverage and bug finding ability, certainly for the scale of systems we have experimented with so far.

3.4 Reliability

For a tester there is nothing worse than trying to use an unreliable test tool. We expect the system we are testing to be unreliable to a certain extent, otherwise we would have no need to test it. But if the test tool is unreliable, then we have to deal with two sources of errors and be able to differentiate between the two causes. Using an open source language allows RBT to achieve modelling reliability for low effort.

4. MODELS AND ORACLES

The first major component of a model-based testing system is the test oracle, which answers the question, "what are the expected results of this test?". If any amount of randomness is used in the selection of test data, the lack of an automatic test oracle would create the serious problem of having to check all test results manually. The oracle can be divided further into two component processes, a **result generator** and a **comparator**[2]. The comparator can translate results from an imperfect results generator, or cope with non-determinism where the results generator predicts more than one possible outcome.

Building the *perfect* oracle, one that can predict the correct answers under all circumstances is both impossible and undesirable. To do so would take comparable time and resources to building the system under test (SUT) and could take the place of that system. From that point of view it is imperative for us to be able to make compromises in building or obtaining oracles. It is always possible to omit some complex system behaviour from an oracle without sacrificing its usefulness; some error handling can often be treated this way. For example, a database system may be designed to cope with disk full errors without losing committed data. The oracle does not have to contain code to cope with the disk full condition, merely be able to check that the SUT does not lose data. For a system which can store persistent state to disk on shutdown and restore its state on restart, the oracle can omit the persistence behaviour and check that before and after states are the same.

Binder[2] p. 924 classifies oracle patterns into four major categories.

Judging. Non-automatic human judgement.

Prespecification. The expected results are calculated prior to the test being run. The patterns in this group are: Solved Example, Simulation, Approximation and Parametric.

Gold Standard. An existing trusted implementation - one that has been used in production: Trusted System, Parallel Test, Regression Test and Voting.

Organic. One relying on approximation or some 'informational redundancy' in the SUT: Smoke Test, Reversing, Built-in Check, Built-in Test, Executable Specification, Generated Implementation and Different but Equivalent.

Table 1 taken from[2] classifies each of the oracle patterns by *fidelity*, *generality* and *cost*. Fidelity is the accuracy of results prediction while generality is the scope of application. The best oracle would be the one with the highest fidelity, broadest range and lowest cost.

With RBT we have successfully used oracles in all categories except the first manual one. It is an important factor in the success of a model-based testing system, that the cost of obtaining or creating an oracle can be minimized. The following sections discuss the variety of approaches we have taken in RBT.

4.1 Smoke Test

The 'smoke test' is one in which the only unexpected results are obviously in error, and that depends on the system being tested. For a graphical user interface, the program ending because of an uncaught exception is obviously an error. For a Java API call, null pointer exceptions are nearly always an error. For those cases where they are not, this default assumption can be changed.

In the case of Java we can also say that any exception thrown which is not contained in the throwables list of a method is also an error. The error could be that the exception is not listed when it should be, or that the exception is thrown in error. The automated test is unable to decide and just notes the discrepancy.

These levels of checking are carried out entirely automatically from the information provided in the Javadoc (or compiled class files or Java source). In this case the oracle is lowest cost - with the appropriate tools, i.e. RBT, near zero cost. By using extra information afforded to us by the nature of the language, we extend the scope of an entirely automatic smoke test.

4.2 Trusted System

Also at the lowest level of cost is the ability to use a trusted system as an oracle. An implementation of a standard, such as the Java Messaging Service (JMS) API, is likely to have other, older and previously tested implementations which can serve as the oracle. There may even be a reference implementation which is probably the ideal case. In the case of regression testing of a system which has had new features added, the old function can be tested by using a previous release as the oracle. An oracle for the new function would have to be derived in an another way.

In both these cases, whether using an alternative implementation or a previous release as an oracle, neither will be perfect; there will be bugs. As the previous release has presumably been acceptable to customers in the past, and satisfied their needs, it will not matter if this oracle causes those bugs to be overlooked or reproduced. In the case of an alternative implementation, it is unlikely that identical bugs will be introduced into both oracle and SUT, so that bugs in both will show up as discrepancies. We are currently using an alternative implementation oracle to test a new JMS provider.

4.3 Simulation

Creating an alternative implementation for the purpose of serving as a test oracle is a relatively high-cost option, and one which has often been viewed as too expensive. This would be the case if this simulation oracle had to adhere to all the requirements of the SUT. But we have successfully written simulation oracles in Python corresponding to Java APIs at a productivity rate that is better than writing traditional test suites. We have also written oracles in Java and a mixture of Java and Python. There is no reason why other languages can not be used, apart from any work needed to interface them to RBT.

The ability to successfully write oracles in a timely manner is due to several reasons:

Simplicity. The oracle implementations leave out many details which are unnecessary to test, or can be avoided (as described earlier).

Performance. This is not a performance test. The oracle is used for functional testing and does not need to run as fast as the SUT. Parallel testing, in which oracle and SUT are run side-by-side, is the slowest as results checking is done on-the-fly as well. Prespecification, in which JUnit test cases are generated, is faster in execution as much of the results checking work has already been done, and output as simpler assertions.

Language. For those written in Python, the language is generally agreed by various informal studies to be significantly (some say 3 to 5 times) more productive than Java.

Many of the features of Python which make it more productive than Java are shared by other languages, such as Ruby. They include interpretation, dynamic typing, easy to use and comprehensive data structures, first-class functions, object orientation, functional programming constructs, modules and packages.

4.4 Hybrid

This pattern is one of our own invention, characterized by high fidelity, broad range and intermediate cost. If a trusted system is at hand which is a close match to the oracle behaviour needed but varies in some of the details, we can write a translation layer which sits over the trusted system and alters its behaviour. We end up with a hybrid of simulation and trusted system oracle patterns, hence the name.

In one instance (Fast Collections), we tested a reimplementation of one of the Java collection classes which was optimized to handle integers only. The interface had also been

Table 1: Oracles Ranked by Fidelity, Generality, and Cost [2]

	Lower Cost		Higher Cost	
	Narrow Range	**Broad Range**	**Narrow Range**	**Broad Range**
High Fidelity	Smoke Test Reversing	Trusted System Regression Testing Parallel Testing	Generated Implementation Different but Equivalent	Solved Example Voting
Low Fidelity	Built-in Test	Judging	Approximation Simulation Executable Specification	

changed to take integer parameters specifically, so a small Python layer was written to make the standard Java collection class match the design of the new class. This layer could have been written in any language which runs in the Java Virtual Machine (JVM).

4.5 Incremental

This pattern is also one of our own invention, characterized by varying fidelity, range and cost. Sometimes it may be convenient to begin with a low cost oracle, so that testing can start immediately. It is useful then to be able to add details to that oracle incrementally, so that gradually it becomes more specific about the results it predicts, while still being operational at every intermediate stage.

We aim to achieve this goal by encoding the smoke test described above as a set of Design by Contract postconditions on each method. If a method returns a value, its postcondition is simply 'true', meaning that no checking takes place. Exceptional postconditions handle the cases where exceptions are raised. For methods that can throw exceptions, the exceptional postcondition states that the only acceptable postconditions are those in the throwables list.

Both types of postconditions can be strengthened as more specific results predictions are added to the oracle. The strongest postcondition is one that states that the result of a method, the value returned, must be a specific value. Intermediate strength postconditions could state that any one of a set or range of values is acceptable. In RBT we already have the capability of following this pattern, but have not yet experimented with it.

4.6 Parallel vs. Prespecification

Any of the previously described oracle patterns can be used in parallel or prespecification fashion. In parallel mode, the oracle and SUT are run at the same time, the test inputs fed to each and the results compared before moving on to the next test inputs. Prespecification in our case means using the oracle to generate Java test programs to fit into the JUnit framework. Both test inputs and expected outputs are encoded therein.

The ability to use the oracles in either way is valuable. The parallel mode is useful for investigative work, and is the default in RBT. The prespecification method allows RBT to produce a standalone, repeatable test that can be given to a developer to recreate a defect. Creating JUnit test programs also demonstrates to the sceptical that this really is test generation, and fits more closely with traditional testing methods and measurements. This can help to lessen any unease felt by testers with model-based testing.

5. TEST SELECTION

Given a suitable test oracle for a system a great deal of the remaining test effort can be automated. In this section we will discuss how our tool automatically generates:

Test Sequences. The actions i.e. method and constructor calls that the test takes against the system and in what order.

Input Data. The data used as input parameters to those actions.

These test selections can then be used to automatically execute tests and determine if they pass of fail using the test oracle.

5.1 Test Data Generation

Essentially we use the Category-partition method[9] to generate test data. Every parameter in the system's API is represented by a class that models the partitions of that parameter. At run time when a parameter needs to be used, i.e. as part of a method call, a value is chosen from the parameter's partitions. The choice is made according to the input coverage criteria described below.

Java basic types are given sensible default partitions but these can be customized on a parameter basis. For example, an integer type will by default have values chosen from the set:

```
{java.lang.Integer.MIN_VALUE,
 java.lang.Integer.MAX_VALUE, -1, 0, 1}
```

If these defaults do not suit then they can be modified by specifying an enumeration of values, a range of values, or indeed both.

5.1.1 Coverage metrics

We define and make use of several coverage metrics in order to drive data selection. These are:

Input Coverage The percentage of possible input values that have been used.

Output Coverage The percentage of possible output values that have been seen.

Data Coverage The combination of input and output coverage.

These percentages must be measured against some metric. The one that is currently used is the *each-choice-used* criterion as defined by Ammann and Offutt[1] in which each possible choice is used at least once. When selecting the next value to use for any parameter we choose randomly from the list of possibilities that have not yet been chosen and choose entirely randomly only when all the input choices have been used at least once. While we have found *each-choice-used* to be surprisingly useful we intend to experiment further with stronger measures such as *base-choice-coverage*[1], *all-pairs* in which all combinations of possible pairs are used or even *all-values* in which all possible combinations of input are tried (though this is not feasible except for very tightly constrained systems).

We believe that a high figure for data coverage during a test run will tend to correspond to a high figure of decision code coverage. So far we have driven data selection based only on reaching 100% input coverage and as can be seen in section 6, high levels of input coverage alone already show promising results. However in future we will investigate methods for choosing input data that covers as much of the output space as possible.

5.2 Test Sequence Generation

Work has been done on test sequence generation[7, 8] using finite state machines (FSMs) which encompass both the possible sequencing and results predictor for a system. Instead we build a separate results predictor, (section 4) and generate sequences based on the systems API. It is our hypothesis, to be investigated, that a test with high levels of input and output coverage using our method may be just as effective as one with high levels of state transition coverage. Furthermore, we propose that building oracles using the techniques we have described is both simpler than building an FSM and that generating tests which reach high levels of input and output coverage will take less time than exploring the full state space of an FSM. Kim and Song[7] define two types of errors that can be discovered via the FSM approach, namely:

Output Fault For the corresponding state and received input the implementation provides an output different from the one provided by the FSM's output function.

Transfer Fault For the corresponding state and received input the implementation enters a different state than specified by the FSM's next-state function.

In practice most transfer faults will be visible by an output fault at a later stage and more importantly building the results predictor as an FSM is only really applicable when the system itself has been specified as a formal FSM. Since this is not normally the case we believe that our approach is much more amenable to testing typical object-oriented systems.

Indeed, object-oriented languages such as Java prove to be advantageous when trying to generate sequences of actions from a system's API:

- An API defined in machine readable format such as that produced by a tool like Javadoc allows us to automatically build a list of all the possible operations in the system. Each of these operations will form a *rule* as described in section 2.
- The valid actions or *rules* at any given time can be automatically inferred from the current state. For example, an object's methods can not be called until an instance of that object exists and a method that has an object as a parameter cannot be called until at least one instance of the required object has been instantiated.

With these two facts alone we can generate random test sequences against the system. At each step a valid *rule* is executed and its results may alter the set of *rules* available for execution at the next step. The selection of the next *rule* is guided by trying to choose input data values to match the *each-choice-used* coverage criterion as described in section 5.1.1.

A key difference between our Rule Based Testing approach and the FSM style is that the internal state of the RBT tool at run time is not a precise state or node of some formally defined FSM. Rather the state comprises the knowledge of which objects are currently instantiated, the possible inputs for each parameter and those inputs that have already been used.

5.2.1 Usage Models

Purely random testing has proved to be thorough for small systems under test and indeed is very useful because it tends to try legitimate paths through a system that a manual tester would not think of, thereby finding different bugs. However, there are reasons why you may want to constrain the test sequence selection choices. For instance, while testing unexpected sequences of actions is useful for testing the robustness of the system and can find many bugs, those bugs are mostly irrelevant if no user will ever use the system in that way. Therefore it makes sense to mostly test the *expected* paths through the system (while keeping an element of pure randomness as users will always use a system in unexpected ways).

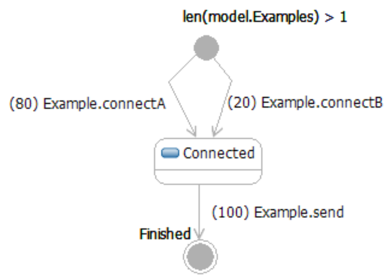


Figure 1: An Example Usage Model

We support this notion with the concept of usage models[10] which in our tool are essentially directed graphs where the nodes are abstract 'states' chosen by the test designer, such as the 'connected' state in figure 1, and the edges are Rules (operations) with a weighting. We call the nodes 'abstract' because they are created and named manually by the test designer as an aid to understanding the flow of execution in the usage model and are therefore not directly mapped to the underlying state of the system. At each step there must be at least one Rule denoted by an edge from the current node that it is executable given the current state.

In the example of figure 1 the usage model will be entered after an instance of the 'Example' Class has been instantiated and it will call the 'connectA' method with 80% chance and the 'connectB' method 20% of the time. By giving only a single edge from a node it is possible to guarantee that that action will be taken allowing you to handle APIs which require particular sequences of method calls. We also support several other features such as 'guarding' out states based on preconditions, selecting particular instances of a Class to call methods on and specifying exact parameters thereby overriding the data selection of section 5.1.

6. RESULTS

The Rule Based Testing tool is being used with considerable success by several teams at IBM, Hursley Park. We present here some preliminary results showing the efficiency of the model based approach versus JUnit test suites in terms of programming effort and code coverage for two APIs. Programming effort was measured in terms of the number of lines of code[1] which while crude is at least measurable. Figure 2 shows the lines of code for the two APIs themselves, their JUnit test suites and the oracle used by Rule-Based Testing.

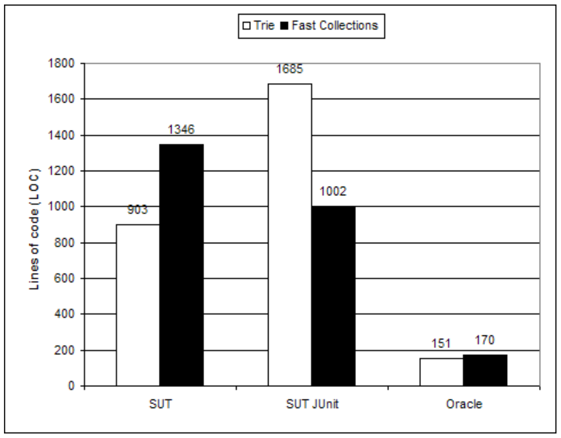


Figure 2: Programming Effort for Implementing and Testing Two APIs

Both APIs are examples of data structures used within larger systems. The Trie oracle was implemented using the *simulation* approach of section 4.3 and the Fast Collections oracle was a *hybrid* and is described in section 4.4. The test oracles accounted for just 6% and 7% of the total effort respectively. In fact, the Trie was an example of a system in which the JUnit test suite was almost twice as large as the system itself. Beyond just lines of code, productivity has been very impressive and the experience of using oracles and the RBT tool very rewarding. In the case of the Trie we were able to build a working model and report defects to the developer within half an hour of receiving the code. Furthermore, as the complexity of the systems increased to achieve higher performance and add features, no changes were required to the existing, already fully functional oracles.

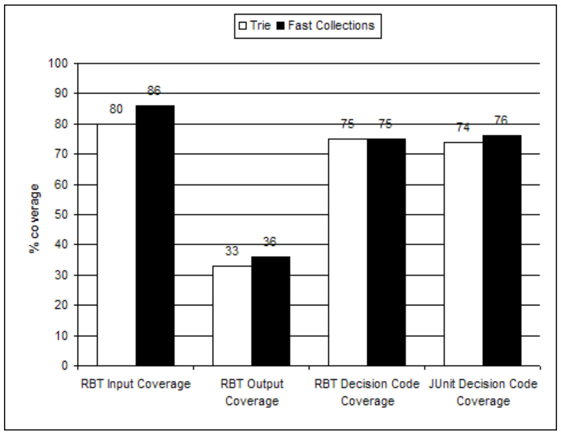


Figure 3: Resulting Coverage

Given such a small amount of relative effort you might expect the JUnit suites to achieve much higher code coverage but this was not the case. In fact, as can be seen in figure 3 the results were very similar. RBT was run over the two APIs without any usage models constraining sequence selection for a few minutes each and the figures also show the amount of input and output coverage that were reached. As can be inferred from figure 3, data selection currently focuses on input coverage and while there appears to be a positive correlation between input coverage and code coverage we hope to investigate the relationship between data coverage and code coverage further (see section 5.1.1).

Of course a high percentage of code coverage does not necessarily correspond to effective testing but unfortunately the real effectiveness of testing is harder to measure. Anecdotally we can say that for these two APIs the use of model based testing found around 30 defects beyond what were discovered through unit testing. Moreover we have other anecdotal evidence from situations were systems had exited functional testing using 'traditional' means but additional testing using RBT found further significant defects. We therefore believe that test suites developed using model based techniques not only take less time to develop but may also be more effective. In the future we will investigate this further by testing systems that have existing standard test suites as described in section 3.

7. CONCLUSIONS

The wide spread adoption of model based testing techniques for object-oriented systems is both feasible and desirable given the advantages offered both in productivity and test effectiveness.

[1]All lines of code figures were generated using David A. Wheeler's 'SLOCCount' program.

The main inhibitors are:

- The difficulty of building oracles.
- The inefficiency of some forms of test generation.
- The lack of robust tools.
- A general reluctance for people to change processes that already work 'well enough'.
- The difficulty for project managers to relate to methods which do not fit their existing methods for measuring progress.

We are attempting to address these concerns by:

- Developing a useful, general purpose method which automates existing test techniques.
- Developing specific tools (for Java APIs) which enables users to get up to speed quickly.
- Spreading the word and using concrete examples to show the effectiveness of the approach.
- Working with management to come up with ways to measure test progress, which are both more relevant to this approach and more generally useful than figures such as the number of successful test cases.

8. ACKNOWLEDGEMENTS

IBM is a registered trademark of International Business Machines Corporation. Java is a trademark of Sun Microsystems, Inc. Python is a trademark of Python Software Foundation. Other company, product or services names may be trademarks or service marks of others.

9. REFERENCES

[1] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 69–80, Gaithersburg, MD, 1994. National Institute of Standards and Technology.

[2] R. V. Binder. *Testing Object-Oriented Systems.* Addison-Wesley, 2000.

[3] I. Craggs, M. Sardis, and T. Heuillard. AGEDIS case studies: Model-based testing in industry. *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 106–117, 2003.

[4] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1), 2002.

[5] R. L. Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley, 2003.

[6] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing.* Wiley, 2002.

[7] C. Kim and J. Song. Test sequence generation methods for protocol conformance testing. *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International*, pages 169–174, 1994.

[8] T. Mori, H. Otsuka, N. Funabiki, A. Nakata, and T. Higashino. A test sequence generation method for communication protocols using the SAT algorithm. *Systems and Computers in Japan*, 34(11):20–29, 2003.

[9] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.

[10] G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Softw. Pract. Exper.*, 25(1):97–108, 1995.

Model-Based Testing for Multi-Language APIs

Rajini Sivaram
IBM
IBM Hursley Labs
Winchester SO21 2JN
+44 1962 817079
rsivaram@uk.ibm.com

ABSTRACT

The increasing complexity of software systems, the shortening development and release cycles and the demands for higher responsiveness and adaptability to changing requirements has made software testing a huge challenge. Even though model-based testing has demonstrated the potential to improve test coverage while reducing test effort, the adoption of these techniques in the industry has been very slow.

This paper presents a model-based testing framework for testing multi-language object-oriented APIs. UML structural diagrams used during model-based design and development are combined with UML behavioural diagrams to build a semi-formal testable specification of the API. Experiences from a case study of the functional verification of messaging clients in four languages are presented to illustrate the benefits of model-based testing. Benefits include improved consistency and interoperability across the messaging client implementations and identification of many specification related defects as well as a significant reduction in the total cost of testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing Tools.*

General Terms

Design, Reliability, Languages, Verification.

Keywords

Model-based testing, model-driven architecture, object-oriented design, UML, Java, .Net.

1. INTRODUCTION

Model-driven design and development are being increasingly adopted in mainstream software development as model transformation and code generation tools are becoming more robust and widely available. Models describe a system from a particular perspective and visual models of complex software systems are often used during design to describe different aspects of the system. The evolution of standards like UML have made modeling of object oriented systems an integral part of object oriented design.

Conventional software testing has so far relied on mental models of the software. Even though model-based testing (MBT) has shown potential to improve test quality and reduce testing effort, MBT is not in widespread use in the industry today. MBT has been used to test many safety-critical systems where the overhead of building formal specifications of the systems is perceived as justifiable compared to the benefits of MBT, particularly measurable and improved test coverage. But the move towards MBT in mainstream commercial software has been slow due to the overhead associated with the move. The lack of easy-to-use tools, the steep learning curve and the difficulty of building formal and complete models have contributed to the slow adoption of MBT.

This paper describes a model-based test framework for object-oriented APIs which closely resembles the approach traditionally used by software testers. This enables a non-disruptive move towards model-driven development and testing at a very low cost. Semi-formal models based on UML activity and sequence diagrams are used to generate readable test code which can be run using existing test automation frameworks, enabling an incremental move towards MBT. A case study is used to demonstrate the advantages of using this MBT approach, especially for multi-language APIs. Improved consistency and interoperability testing across multiple implementations of the API was one of the primary goals of MBT in the case study.

2. BACKGROUND

Research has shown that software testing based on rigorous formal specification of software systems can achieve extensive and measurable test coverage [7][8]. MBT based on formal methods has been applied to many safety critical systems and in hardware testing. The experiments conducted by Farchi et al. show that behavioural models derived from software specifications can be used in conformance testing and that the generated test suites achieve higher levels of code coverage compared to traditional test suites [6].

Bishop et al. present a technique for rigorous protocol specification that supports formal specification based testing [7]. Formal specification of complex protocols like TCP provide an unambiguous specification that can be directly used in testing the implementations. Offutt and Abdurazik present a technique that adapts pre-defined state-based specification test data generation criteria to generate test cases from UML statecharts [8]. The project attempts to provide a solid foundation for generating tests from system level software specifications of safety critical systems using new coverage criteria. Clarke describes the use of state based behavioural models in testing telecommunications systems and presents the techniques used to overcome the state explosion problem [9]. The comprehensive set of tests generated using path analysis of the models resulted in significant productivity gains. An architecture for model-based verification and testing using a UML profile is presented by Cavarra et al. [10]. Class, objects and state diagrams are used to define the models which can be compiled into a tool language and used to generate tests. Gresi et al. introduce a mathematical foundation for formal conformance testing and automatic test generation based on UML statecharts [11].

While the techniques based on formal state-based semantics have proven to be very effective, the adoption of these techniques for testing commercial software has been very limited. The culture change and skills required to switch over to a revolutionary new testing methodology based on formal methods has limited the use of MBT in mainstream software testing. SeDiTeC is a tool which enables the use of UML sequence diagrams for testing [12]. However issues of concurrency, integration with other UML diagrams and integration with existing test frameworks are not addressed.

The MBT framework described in this paper attempts to overcome the barriers to adopting MBT in mainstream testing by enabling incremental adoption of MBT using an approach that is familiar to most testers. A semi-formal specification of the API using UML behavioural diagrams is used so that existing informal specifications can be easily modeled. This approach is aimed at generating the same types of tests that are hand-crafted by experienced software testers. Unlike state diagrams, sequence and activity diagrams can be transformed to readable code that can be easily understood and debugged by developers who are not familiar with MBT. In addition, tests aim to reuse API models developed during the design process, making it possible to maintain a single source for design, API interfaces, API documentation as well as tests.

Many object oriented languages are currently used to develop applications in different domains, and hence many APIs including the messaging API described in Section 4 have definitions and implementations in different languages. Conventional testing of multi-language APIs have so far relied on manual porting of tests into different languages since automated language translation tools are often inadequate. MBT is particularly suited for testing multi-language APIs since the goal of functional verification of the APIs includes uniform testing and consistency and interoperability tests.

3. MBT USING UML

3.1 Model-based tests

The MBT framework described in this paper is based on semi-formal models in UML and is targeted at testing mainstream commercial software and middleware where the complexity of building formal models and the steep learning curve associated with formal modeling have prevented the use of MBT. The use of visual models of tests combined with the generation of readable code which can be executed using existing test automation harnesses enable the incremental adoption of MBT in traditional test organizations.

UML has been accepted as the standard graphical modeling language across the software industry for specifying, constructing, visualizing and documenting software systems [1]. Model-based design using UML has been used to build blueprints of software systems for nearly a decade [2][3][5]. UML is used to model APIs in many software projects, and these models are often transformed to generate the final interfaces and documentation. The use of these API specification models in testing will ensure that tests are always synchronized with the development or released API versions. This is a particularly significant advantage when using iterative development.

The structural model of an API typically consists of a collection of classes containing operations and relationships which include dependencies as well as class hierarchies. Behavioural diagrams used to describe the dynamic behaviour of the API may be sequence diagrams, activity diagrams or state diagrams, depending on the particular behaviour that is being described. For example, multi-threaded or multi-process behaviour are easily visualizable using activity diagrams while certain state based systems are best described using state diagrams. A testing profile is used to add additional test specific information into the behavioural diagram. Stereotypes provided by this profile enable the definition of test suites, parameter value ranges and other test related information required for the generation and execution of tests. Run-time constraints are added to the behavioural model to define pre- and post-constraints and invariants. The constraints are transformed to assertions in the final test code, and these assertions are used to assign pass or fail verdicts to the tests.

Differences in the APIs introduced due to language conventions can be easily encapsulated in the specification model of the API in such a way that transformations for different languages generate optimal code for each language. A language-specific UML profile is defined for each supported language to add language-specific details to the model. The use of unified models for testing multi-language APIs results in consistent tests across the different implementations while reducing the total cost of testing.

Test data can be specified in the model for data that is specific to the test. Data values can be specified for parameter values, initial values of variables, values used in conditional statements, loops etc. Ranges of values can also be specified, and the transformation can be instructed to generate random or boundary value tests. Test configuration and deployment information is also specified in the model. This information is used to define test suites, the different processes that make up each test, as well as their arguments. The configuration information is used to generate test execution scripts which can be used to run the tests standalone during development and debugging or inside existing test automation frameworks.

3.2 MBT Transformation Framework

The MBT transformation framework has been implemented as an Eclipse Plugin integrated into Rational Software Architect (RSA) [14]. RSA is an integrated design and development tool that leverages model-driven development using UML. RSA is built on the extensible Eclipse platform [17] and provides a model transformation framework that can be used to transform models into other artifacts like a more refined model, code or documentation.

The transformation framework consists of a generic multi-language code transformation plugin which transforms models containing behavioural diagrams into source code. The transformation framework uses extensions to support test related code and data generation and the extensible architecture enables easy integration of other test generation tools. Transformations are currently implemented for Java, C++, C, C# and Managed C++ for .Net. Figure 1 shows the architecture of the multi-language code transformation framework.

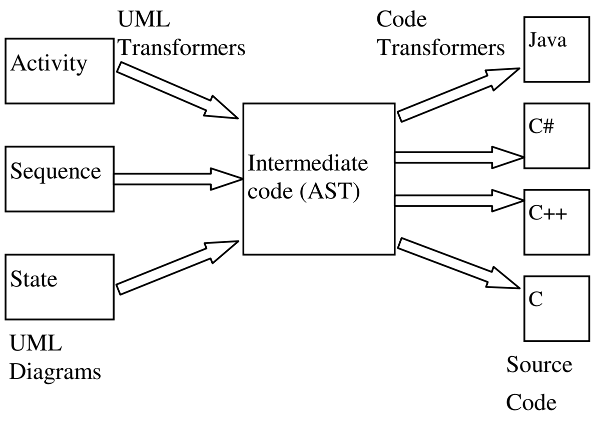


Figure 1. Code Generation Framework

UML behavioural diagrams like sequence diagrams or activity diagrams are first converted into an abstract syntax tree (AST) to provide a common intermediate representation that can be converted to code in different languages. AST is then transformed to code in the chosen language(s).

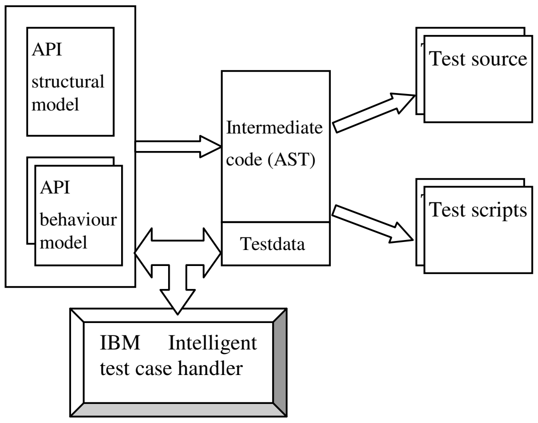


Figure 2. Test Generation Framework

The multi-language code transformer is extended with a test transformer to generate tests from UML behavioural diagrams as shown in Figure 2. The test transformer generates appropriate data sets for tests based on the possible values specified in the model. The transformation architecture is extensible and provides three extension points.

1. UML transformation – Each UML transformation transforms an UML diagram to intermediate code. New transformations can be added to transform other diagrams or provide alternative transformations, for instance for state diagrams.
2. Code transformation – Each code transformer transforms the intermediate code representation to source code in one language. Code can be generated for a new language by adding a new transformer.
3. Existing tools and algorithms which generate test behaviour from models or test data sets can be integrated into the UML transformation. Figure 2 shows the use of one such tool in test data generation.

Combinatorial covering suites have been shown to lower the cost of testing while providing measurable coverage [4]. This approach involves identifying parameters that define the space of possible test scenarios and then selecting the set of test scenarios in such a way as to cover all pairwise (or t-wise) interactions between the parameters and their values. The IBM Intelligent Test Case Handler uses sophisticated combinatorial algorithms to construct test suites with given coverage properties over large parameter spaces [16]. The model-based test framework has been integrated with this test case handler to generate test data sets. Parameters and variables and their interactions as well as values to be used for testing can be specified in the test model. Test suites are generated by the transformation with the specified coverage property.

4. MBT FOR MESSAGING CLIENTS – A CASE STUDY

4.1 Java Message Service and IBM Message Service Clients

The Java Message Service (JMS) API is a messaging standard which enables applications written using the Java 2 Platform Enterprise Edition (J2EE) to send and receive messages using any J2EE-enabled enterprise server [15]. IBM Message Service Clients (informally known as XMS) aim to provide a consistent cross-language API that brings the benefits of the JMS messaging standard to non-Java platforms. XMS is currently implemented for C, C++ and .Net.

Unlike network protocols, JMS has a very flexible messaging specification, making it difficult to define a formal specification of JMS. In addition, the testing in the case study is aimed at four different implementations of JMS/XMS. The flexibility in the specification in exploited in different ways in the different implementations. Formal specification of JMS for these environments using standard state based specifications would require separate models for each of the implementations. In addition, lack of skills in formal methods in the industry makes it very expensive to introduce testing based on formal specifications.

Like most multi-language APIs, conventional testing of JMS and XMS APIs relied on manual porting of tests to different languages. This process has proved to be both expensive and error-prone. Even though consistency and interoperability across the clients is one of the main goals of the messaging standards, these proved to be the biggest challenge for test using conventional testing which relies on test teams skilled in individual programming languages.

Concurrency testing is another area where conventional testing proved hard to implement. Skills in concurrent programming, particularly in languages like C where different code is required to create multi-threaded tests on different platforms and the difficulties in visualizing and debugging concurrent tests restricted the level of concurrent testing.

In this case study, traditional testing for JMS and XMS was performed by two separate test teams, with team members assigned to testing the API for a specific language. This testing has been largely based on the informal JMS specification. MBT was introduced at a later stage with one developer assigned to develop the test framework and the models for specification and testing. The teams were not skilled in UML and UML modeling was introduced into development and test for the first time as part of this effort.

4.2 MBT of JMS and XMS

MBT of JMS and XMS is based on a unified specification of JMS and XMS APIs in different languages. The structural model of the API describes the interfaces in the different languages using the language-specific profiles. A semi-formal behavioural specification consisting of activity diagrams and sequence diagrams based on the informal JMS specification is used for test generation. Unified testing of the clients aims to ensure that the different implementations of the clients are consistent and interoperable. Behavioural differences in implementations introduced due to language differences and differences in transport are highlighted in the test models, and the transformation framework generates different tests in these cases.

As an example consider the message ordering tests for JMS.

JMS defines that messages sent by a session to a destination must be received in the order in which they were sent. This defines a partial ordering constraint on a session's input message stream [15].

Figure 3 shows an activity diagram which sets the context for the test. It consists of two partitions, one for the producer which sends messages and another for the consumer. Figure 4 shows a simple sequence diagram which represents the consumer which receives messages and it shows the constraint about message ordering. The basic test generated in each language from these two diagrams creates two threads, and the test succeeds if the messages are received in the consumer thread in the same order as they were sent by the producer thread. Variations of this test can include

1. Same test with different message types
2. Same test with different message sizes
3. Different combinations of message persistence, acknowledge modes of sessions and message priorities can be generated. Combinatorial coverage analysis can be used to generate the minimal set.
4. The fork in the activity diagram can be transformed to generate multiple consumer threads using the same ordering constraint.
5. The same model can be used to generate multi-process tests where the producer and consumer reside in different processes. The resulting multi-process tests can be used to verify that the constraint holds even when the producer and consumer are in different languages.
6. Additional run-time constraints can be placed to ensure that all messages are received and in the correct order. The constraint can have a special case for high-performance implementations where non-persistent messages are not guaranteed to be delivered.
7. Large numbers of threads, large loop sizes and large message sizes can be used to generate stress tests using the same model.

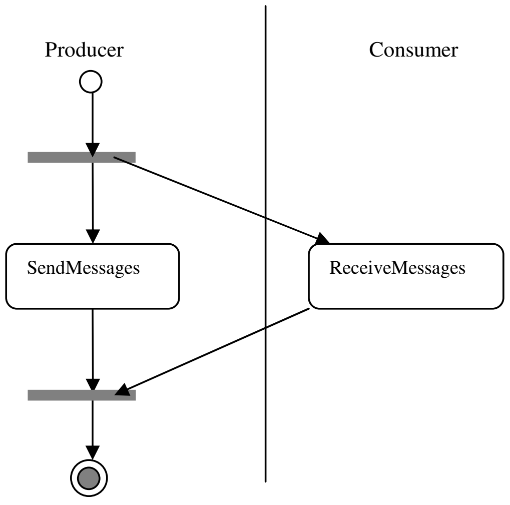


Figure 3. Activity Diagram

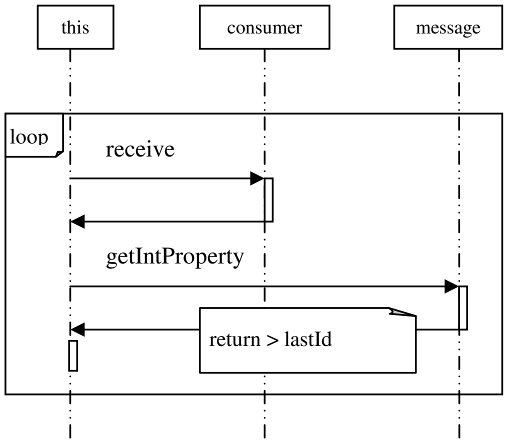


Figure 4. Sequence Diagram

```
public void receiveMessages(MessageConsumer
consumer, int messageCount) throws Exception {
  int lastId = 0;
  for (int i = 0; i < messageCount; i++) {
    Message message = null;
    int thisId = 0;
    message = consumer.receive(1000);
    thisId = message.getIntProperty("sequence");
    Assert.isTrue(thisId > lastId, "Constraint
failed: return > lastId");
    lastId = thisId;
  }
}
```

Figure 5. Generated Java Code

Figure 5 shows the snippet of Java code generated from the sequence diagram in Figure 4 when logging and exception checking are turned off. When verbose logging is enabled, the generated code contains extensive logging which log method calls, parameters, return values and exceptions with the source line numbers.

4.3 Advantages of MBT

4.3.1 Test quality

Model-based tests are currently implemented for JMS/XMS for Java, C, C++ and .Net for messaging clients running against three different enterprise messaging servers. Model-based tests for JMS and XMS have uncovered more than 80 defects related to the specification, consistency and interoperability across the clients, which were not discovered by the hand-crafted testing effort. Unified specification and testing for the different clients, the ease of generation of concurrency and interoperability tests and the improved behavioural specification of the API based on a visual semi-formal model rather than textual informal prose have contributed to the better quality of testing. Better test data generation algorithms are expected to improve this even further.

A large number of defects found using MBT which were not discovered using hand-crafted testing were related to the interpretation of the specification of JMS. In the multi-language scenario, the discovery of the differences across the different implementations as a result of unified testing enabled the development of a cross-client behavioural specification, which is more rigorous than the informal JMS specification. Improving the clarity of specifications using visual behavioural models will also benefit APIs which are targeted at a single language.

4.3.2 Test resource

Figure 6 shows the total test team size for functional verification of JMS and XMS, including the additional testing of new clients currently under development. The initial cost of implementing the model-based test framework and skill development for MBT are also included in the figures and the results demonstrate that the cost of deploying MBT can be easily recovered. The figures include estimates covering the period till the end of 2006.

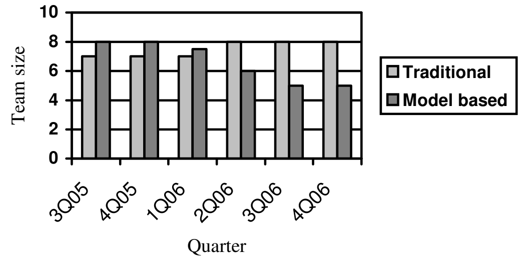


Figure 6. Test Team Size

Figure 7 shows the relative code size of the test code used for functional verification of JMS and XMS. The traditional code size refers to actual code size in kilo lines of code for existing code and estimated code size for testing the new client. MBT code size refers to the equivalent lines of hand written code that the model describes. The significant reduction in test code size, and consequently in test maintenance is achieved using MBT because of the use of the same tests for different languages and different versions of the messaging clients. The transformation plugin source (20 kloc) is not included in the MBT code size since the plugin is generic and can be potentially used with to test multiple APIs.

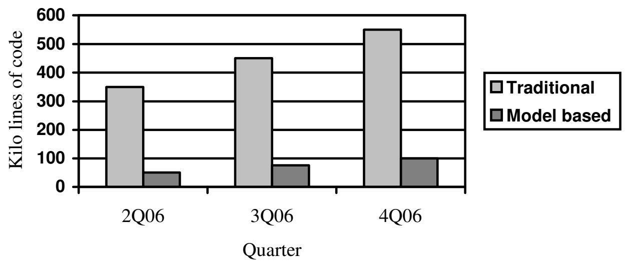


Figure 7. Estimated Test Code Size.

Figure 8 shows the total test analysis effort in person months. The total test analysis remains roughly the same with MBT. The slight increase in the initial stages is a result of higher number of tests being run and the slight reduction in the later stages results from increasing consistency of tests and test results.

Figure 9 shows the relative cost of developing new tests using model-based and traditional methods. There is a significant reduction in cost for adding new tests even when the additional cost of skill development for MBT is taken into account since the same tests are run against four languages and different versions of the API.

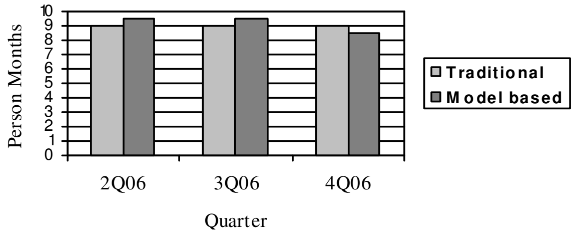


Figure 8. Estimated Test Analysis Effort

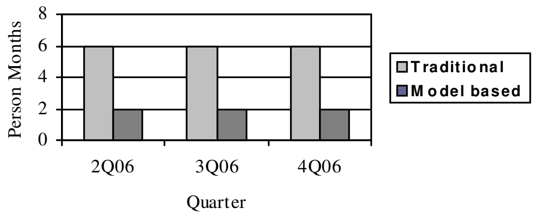


Figure 9. Estimated Test Development Effort.

Figure 10 shows the relative cost of testing messaging clients for a new language. Traditional testing cost is based on the person months used to implement functional verification tests for XMS .Net using C#. MBT cost is based on the implementation of the transformation for generating C# code and the specification model for .Net.

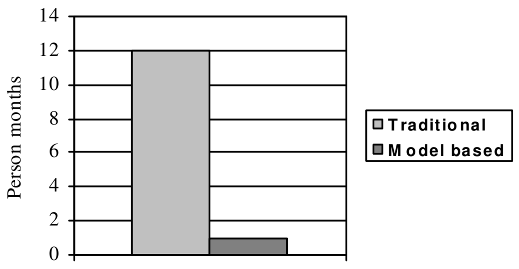


Figure 10. Addition of tests for a new language.

4.3.3 Model reusability

Unified modeling of JMS and XMS has enabled a consistent definition of the API which can be used to generate interfaces in different languages as well as the documentation. The same models are used for functional verification. These models have been further used in unit testing the new IBM JMS client implementation using mock objects. The API specification models have also been used in conjunction with behavioural models of performance test scenarios to generate performance tests for messaging clients. The separation of test data from test behaviour, the ease of modeling and visualizing concurrent tests, and the use of a common API specification model have contributed to fast turnaround of performance tests.

4.3.4 Other advantages of MBT

MBT enables the separation of test behaviour from test data, enabling the generation of large numbers of tests from a small number of models. Existing combinatorial algorithms are used to generate minimal test suites with strong coverage properties.

MBT enables language differences in APIs to be encapsulated into the API specification model so that the same behavioural specification of the API can be used to generate tests for different languages, or even different versions of the API. Object-oriented APIs implemented in non-object-oriented languages like C can also be handled if the API definition is consistent.

Multi-process tests can be generated as multi-threaded tests for initial development and debugging. Tests can be generated with extensive logging or with customized selective logging. Tests can be generated to run within an automation harness or as standalone tests for debugging. Stress tests using large numbers of threads, long loops or large arrays can be easily generated from the same test models.

The incremental adoption of MBT and the generation of readable code that is very similar to the code that is usually hand-crafted by experienced testers has made it practical to introduce models into the software development and testing process without causing any disruption. The success of MBT in functional verification has also encouraged the use of modeling during the design stage in the development of a new JMS client. The extensible architecture of the MBT framework will enable the integration of formal state-based testing in future to test critical sections of the code.

The visual modeling of API behaviour promotes better understanding of the API specification amongst both developers and testers, resulting in early identification of defects. UML state diagrams enable the precise modeling of parts of the API which are formally specified. Sequence and activity diagrams enable the development of models that are easily understood and provide the flexibility to model parts of the API which have incomplete definition in order to allow high-performance flexible implementations.

The use of models developed during the design process for testing enables tests to be generated even before the code is implemented. Combined with mock objects used in unit testing, generated tests can be used very early in the development cycle to discover not only implementation defects, but also specification and requirements defects.

While MBT based on UML behavioural models is well suited to functional verification tests, scenario and system tests require models which can combine multiple behavioural models in different ways. Patterns for messaging clients enable the development of patterns and pattern transformations which can be used not only for system tests, but also for the generation of messaging application code [13]. These transformations also benefit from multi-language code generation, and in addition enable the generation of deployment scripts for different messaging servers. System test transformations for messaging clients using patterns are currently being developed.

5. SUMMARY AND FUTURE WORK

This paper explores the use of semi-formal models for MBT to enable the incremental adoption of MBT in test organizations which are currently reluctant to adopt MBT due to the high overheads and lack of skills in formal modeling. The case study illustrates that the cost of implementing a MBT framework and skill development required for the implementation of MBT can be easily recovered in a single project, especially when testing multiple languages. Most of the results presented in this paper are equally valid for code targeted at a single language, even though the cost benefits may not be so dramatic.

Visual modeling and the use of a common set of models throughout the software development lifecycle promote better understanding of the software and early identification of defects. The use of a familiar test paradigm and the generation of readable code which can be executed within existing test automation frameworks lower the initial investment in the adoption of MBT. The extensible architecture used by the test generation framework enables existing model-based code and data generation tools to be easily integrated into the framework.

Scenario and system test generation using the model-based test framework and patterns for messaging clients are now being developed to support model-based system testing. Future enhancements to the toolset include limited round-trip engineering capability from Java to enable existing tests to be converted to models easily. Integration of the framework with tools supporting formal state-based specifications is also being explored.

6. ACKNOWLEDGMENTS

I would like to thank John Kinchen and Keith Watson for funding this project. I would also like to thank Alan Hartman, Mila Keren and Sergey Olvovsky from IBM Haifa Research Labs for their help in integrating IBM Intelligent Test Case Handler with the model transformation plugin.

7. REFERENCES

[1] Object Management Group. *Unified Modeling Language SuperStructure Specification, version 2.0.* Revised Final Adopted Specification (ptc/04-10-02), 2004.

[2] Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide.* Addison Wesley, 2005.

[3] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual.* Addison Wesley, 2004.

[4] Hartman, A. Software and Hardware Testing Using Combinatorial Covering Suites. In *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, Kluwer Academic Publishers, 2003.

[5] Objects By Design. *UML Products by Company*. http://www.objectsbydesign.com/tools/umltools_byCompany.html..

[6] Farchi, E., Hartman, A. and Pinter S.S. Using a model-based test generator to test for standards conformance. *IBM Systems Journal 41*, 2002.

[7] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M. and Wansbrough, K. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of ACM SIGCOMM* (Aug. 2005).

[8] Offutt, J. and Abdurazik, A. Generating Tests from UML Specification. *Second International Conference on the Unified Modeling Language (UML99),* Springer-Verlag, New York, 1999.

[9] Clarke, J. Automated Test Generation from a Behavioral Model. In *Proceedings of Software Quality Week conference,* May 1998.

[10] Cavarra, A., Chrichton, C., Davies, J., Hartman, A., Jeron, T., Mounier, L. Using UML for Automatic Test Generation, In *Tools and Algorithms for the Construction and Analysis of Systems*, (Sept. 2000), Oxford University Computing Laboratory.

[11] Gresi, S., Latella, D., Massink, M. Formal Test Case Generation for UML State-Charts, In *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems* (2004).

[12] Fraikin, F., Leonhardt, T. SeDiTeC - Testing Based on Sequence Diagrams. In *IEEE Intl. Conference on Automated Software Engineering* (2002).

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[14] Quatrani, T., Palistrant, J. *Visual Modeling with Rational Software Architect and UML.* Addison Wesley, 2006

[15] Sun Microsystems, Inc. *Java Message Service.* Version 1.1 (April 2002).

[16] IBM Corporation. *IBM Intelligent Test Case Handler.* http://www.alphaworks.ibm.com/tech/whitch

[17] Object Technology International, Inc. *Eclipse Platform Technical Overview*. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

Semi-Automatic Test Case Generation from CO-OPN Specifications[*]

Levi Lúcio
Software Modeling and Verification laboratory
University of Geneva, Switzerland
levi.lucio@cui.unige.ch

Didier Buchs
Software Modeling and Verification laboratory
University of Geneva, Switzerland
didier.buchs@cui.unige.ch

Luis Pedro
Software Modeling and Verification laboratory
University of Geneva, Switzerland
luis.pedro@cui.unige.ch

ABSTRACT

In this paper we will describe our work on automatic generation of tests for Systems Under Test (SUT) described in the CO-OPN specification language that has been developed in our laboratory. CO-OPN (Concurrent Object Oriented Petri Nets) is a formalism for describing concurrent software systems that possesses most of the characteristics we can find in mainstream semi-formal modeling languages. Given its formal semantics, CO-OPN is a suitable formalism for model-based test case generation.

Within our work we have developed a test selection language for CO-OPN specifications which we have named SATEL (Semi-Automatic Testing Language). The purpose of SATEL is to allow the test engineer to express abstract *test intentions* that reflect his/her knowledge about the semantics of the SUT. Test intentions are expressed by execution traces with variables that can expanded into full test cases – this is done using the CO-OPN specification of the SUT as a reference for calculating the oracles. We call our test selection language *semi-automatic* because it allows explicitly choosing the shape of the test cases in a test intention, while relying on automatic mechanisms for calculating the equivalence classes of operations defined in the model.

1. INTRODUCTION

Many are the difficulties that arise when one tries to perform automatic test generation for a software system. Let us start by defining what is meant by "automatic". The approaches and tools that exist today for performing test generation can be mostly divided into two categories: white-box and black-box. While white-box testing is usually based on covering certain execution paths of the SUT (System Under Test) code, black-box testing is concerned about covering functionalities of the SUT described in a model. In any of these cases the automatic aspect of test generation is made difficult by the fact that the set of reachable states of either the SUT or the model is, in the general case, infinite. In this context, the notion of coverage of an SUT or a model by the generated tests depends strongly on which restrictions were made in order to "intelligently" cover the given state space. For example, if one is performing white-box testing, covering all decision points in the code is a classic strategy [1] and can automatically generate test cases. Using this kind of reasoning, many other strategies for "automatic" white-box test case generation might be devised.

On the other hand, by introducing the notion of a model of the SUT it might be expected that the problem of coverage of the SUT would be implicitly solved. Being that the model is an abstraction of the system we pretend to test, we could assume the abstractions introduced by the model would make extensive testing – modulo those abstractions – possible. In reality this is not true. Models of software systems can be also very complex and mainly abstract from having to deal with hardware, operating systems, software libraries or specific algorithms (unless we are aiming at directly testing those entities). What the model introduces in the chain of test generation is rather a form of redundancy – a way of comparing the SUT with an abstraction of itself in order to find differences which are possibly errors in the code. Again, we meet with the problem of reaching an infinite set of states of the model.

1.1 Our Approach

In this paper we propose a semi-automatic approach to test generation. The approach is semi-automatic in the sense that we allow the test engineer to state *test intentions*, while using unfolding techniques (introduced by Bernot, Gaudel and Marre in [2]) for automatically finding equivalence classes of inputs to operations of the model. It is our intention to explicitly make use of the test engineer's knowledge in the test generation process. He/She will be able to express which parts of the model are relevant for testing and to impose limits on how that testing should be performed. However, given an operation of the model, he/she will also be able to state that the generated tests should include inputs that automatically cover the various behaviors (the equivalence classes) of that operation. For example, while testing a Banking application, the test engineer would be able to express that he/she wants a certain sequence of operations to be executed during testing (e.g. *login user / introduce password / deposit amount / withdrawal amount*), but also to

[*]This project was partially funded by the DICS project of the Hasler foundation, with number 1850.

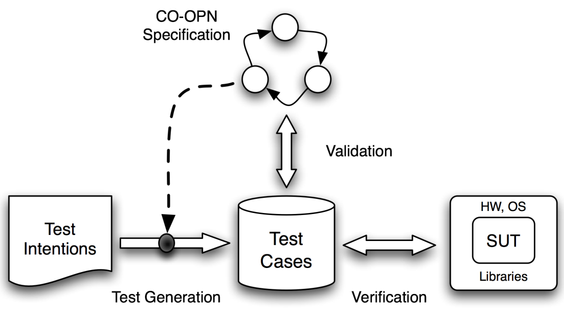


Figure 1: Testing Process

express that the generated tests should automatically cover all the behaviors of the *password* operation – the password is either correct or incorrect for a given user.

Figure 1 depicts the process of testing an SUT using our approach. Assuming a CO-OPN [3] specification of the SUT exists, the test engineer writes a script of test intentions. These test intentions may make use of the semantics of the model in order to automatically cover equivalence classes of specifications' operations (hence the dashed line). The test cases produced by the test intentions are then confronted with the specification for validation purposes. This step is necessary in order to decide whether a test case is a valid or an invalid behavior according to the specification. An example of an invalid behavior would be expecting an incoherent output while applying a given input to an operation of the specification. Although valid behaviors are the most interesting, invalid behaviors can also be used as test cases in the sense that they depict scenarios of execution that should not be allowed by the SUT. The rightmost arrow of figure 1 represents the verification of the SUT by submitting to it the validated test cases. This step is by no means simple and requires the existence of oracles for the generated test sets and drivers actually apply those tests to the SUT. Given that our current research is focused on producing test cases and their oracles, we will not explore the test driver issue in this paper.

1.2 Contribution

The novelty of our approach lies in the fact the we privilege the test engineer's semantical knowledge of the SUT rather than emphasizing pure automatic test generation as for example in [4]. To our knowledge current research on testing is either focused on test drivers without automatic test generation abilities – e.g., the very successful TTCN-3 [5] – or on approaches which aim at building press-button tools. In the latter category we can mention the CLPS-B for limit testing approach from the university of Franche-Comté [4] which tries to automatically reach possible outcomes of the operations of a model expressed in the specification language B. Although very effective in certain situations, this approach implicitly requires a strong discretization of the model so the state space can be searched. Another approach with which we identify more is the AsmL Test Tool from Microsoft [6]. In this tool it is possible to generate a state space for a specification in the AsmL language and the test engineer can provide abstractions both for the state space itself and for input values. Algorithms for searching the reduced state space allow generating test cases that cover all or part of those states. Although the approach is in certain aspects similar to ours, we consider our test selection language to be more appropriate for the test engineer to express semantic knowledge about the SUT at a high level of abstraction. The TOBIAS [7] tool is an automatic test generator based on producing combinatorial test sets from test purposes [8]. The idea of using test purposes is similar to that of using test intentions, although the TOBIAS tool suffers from the problem of calculating the oracles for the test sets. This is due to the fact that no semantically exploitable specification exists – contrarily to our approach. Despite, the authors try to overcome that difficulty by using VDM or JML assertions as a means of filtering interesting test cases [8] from the large combinatorial test sets.

Another aspect of our contribution lies in the fact that we use CO-OPN as our specification language for model-based test case generation. CO-OPN is based on Algebraic Data Types for describing data types and on structured Object Oriented Petri Nets for describing behavior (see [9] for details). The Petri Nets semantics of our specification languages places us very near to such formalisms as Harel's statecharts [10] or the Behavioral State Machines introduced in UML 2.0 [11] (which is in fact an object-based variant of Harel's statecharts). We are currently pursuing our research in the sense of applying our test intention-based techniques to test generation based on such mainstream models. We accomplish this by translating those models into CO-OPN and profiting from the partial equivalence of the semantics, as explained in [12].

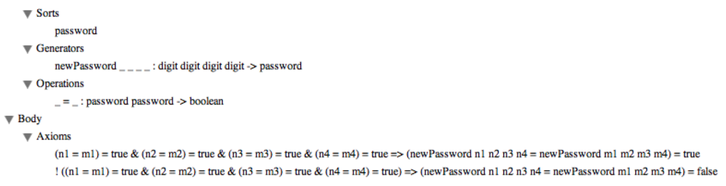
1.3 Organization of the paper

The remaining of this paper is organized as follows: section 2 describes CO-OPN as a modeling language and introduces its semantics by means of an example. Section 3 discusses some of the requirements for a test intention language that will act over CO-OPN specifications. In section 4 we present the syntax and semantics of our test intention language SATEL. Finally, section 5 provides a concrete example of usage of our language and section 6 concludes.

2. THE CO-OPN SPECIFICATION LANGUAGE

In this section we will introduce the CO-OPN specification language. The objective is not to provide an exhaustive definition of the language, but rather to explore issues that are relevant while designing a language for semi-automatic test generation using CO-OPN specifications as models. In order to do this we establish an example of a Banking system model in CO-OPN. This example will be used throughout the paper.

Our Banking system is a multi-user centralized system that provides to users the possibility of managing their accounts via remote automatic teller machines. Before being able to perform operations on his/her account, the user has to authenticate in a two-step process: log in with a username; if the username exists, the system asks the corresponding password. If the user provides three wrong passwords, his/her account will become blocked and he/she will no longer be able to connect to the system. After having successfully authenticated, the operations available to a user are *balance display*, *money deposit* and *money withdrawal*.



```
▼ Sorts
      password
▼ Generators
      newPassword _ _ _ _ : digit digit digit digit -> password
▼ Operations
      _ = _ : password password -> boolean
▼ Body
  ▼ Axioms
      (n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = true
      ! ((n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true) => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = false
```

Figure 2: ADT Password

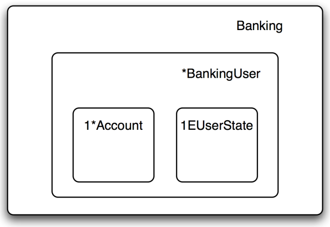


Figure 3: Class Model for the Banking System

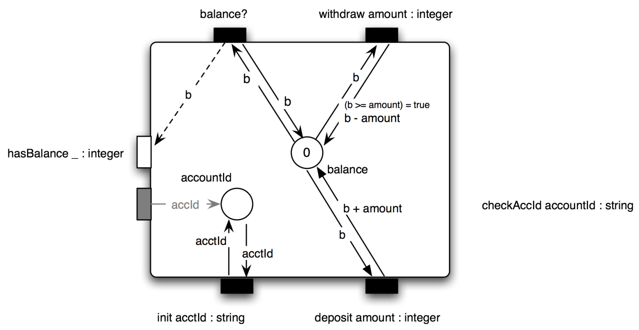


Figure 4: Account Class

The following subsections introduce the different kinds of modules which can be present in a CO-OPN specification: *Algebraic Data Types* (ADT for short) and *Classes*. ADT correspond to data structures defined by algebraic sorts and operations. The equational algebraic definition of the data type's operations allow us to perform the unfolding of the behavior of those operations in a way that is useful for test generation, as we will describe in this paper. CO-OPN's Classes are relatively compatible with the popular notion of Class in the Object Oriented paradigm.

2.1 Algebraic Data Types

ADT are an instance of the well known notion of algebraic specifications (interested readers are directed to [13]). An ADT module includes *generators* and *operations*. Generators build the set of elements (the *sort*) of the ADT, while operations are functions over that set. The behavior of the operations is defined by algebraic equations.

Coming back to the Banking example, we can find in figure 2 a partial definition of an ADT for the Banking specification. The ADT defines the sort *password*, which is the set of passwords the Banking system allows. In the figure it is possible to see that that the single generator of the sort is called *newPassword* and includes four elements of sort *digit* as parameter (the sort *digit* – numbers from 0 to 9 – is defined in a separate ADT module). For the *password* Sort only the equality ("=") operation is defined.

2.2 Classes

In figure 3 we have represented the class model of our Banking system. The diagram models a system that contains zero or multiple instances of class *BankingUser*. A *BankingUser* object holds the user's current state (*not logged / waiting for password reply / logged / blocked*) and a number of accounts. The user's current state is held in one instance of the *eUserState* class – the prefix 1 indicates only one instance exists per instance of *BankingUser*. Each account is an instance of class *Account* and a user can own one or multiple accounts.

2.2.1 Class Interface – Methods and Gates

In CO-OPN an instance of a class is seen as an entity that can require and/or provide events inside a network of objects. Required events are called *methods* and provided events are called *gates*. Both method and gate events may be parameterized.

Figure 4 depicts a graphical representation of a simplified *Account* class of our Banking system. Methods and gates are represented respectively by black and white rectangles on the outside of the class. An object of type Account requires (or is able to respond to) three outside events: *balance?*, *withdraw amount : integer* (meaning *withdraw* takes an amount of money as parameter), *deposit amount : integer* and *checkAccId accId : string*. An account is also able to produce one event to the outside: the *hasBalance _ : integer* gate outputs the amount of money present in the account. In this particular case, the *hasBalance _ : integer* event is produced as a response to the *balance?* one.

The method *init accId : String* corresponds to the initialize method of the class. Although all classes in CO-OPN have an implicit *create* method, in this case *init* is used in order to initialize the identifier of a particular account.

2.2.2 Petri Nets for Behavior

The state of a CO-OPN object is held in a petri net. For readers who are unfamiliar with the formalism, Petri nets are a means for representing the behavior of concurrent systems. In a Petri net two concepts are fundamental: *places* that hold resources and *transitions* that can consume and produce resources. Newly produced resources are again placed on the net. Typically places are represented by circles, transitions by solid rectangles and their interactions by arcs.

A CO-OPN class can be seen as an encapsulation of a petri net with methods and gates. In figure 4 the places of the petri net are represented by the circles labeled *balance* and *accountId*. In this particular case the existing transitions are implicitly associated to methods and gates of the class[1]. For example, the *withdraw amount : integer* method, when activated, takes a resource *bal* (implicitely meaning an inte-

[1]It is also possible to have transitions inside the class which are not associated to methods or gates

ger representing the balance of the account) from the place *balance* and puts it back subtracting the amount that was withdrawn. In order for this event to happen the existing balance has to be superior or equal to the amount of money withdrawn – as the label in arc from the *withdraw* method to the *balance* place indicates. Another interesting example is the *balance?* method that checks the existing balance (without changing it) and activates the gate *hasBalance _ : integer*[2] with the available balance as parameter. We provide in the following lines the textual representation (the so called axioms) of the *withdraw* and *balance?* methods depicted graphically in figure 4.

```
(b >= amount) = true => withdraw amount ::
     balance b -> balance b - amount

balance?  with hasBalance b ::  balance b -> balance b
```

Briefly, the first axiom for the *withdraw* method is split into three parts: the condition *(b >= amount) = true*, the method (and parameter) *withdraw amount* and the necessary pre- and post-conditions of the object's petri net *balance b* → *balance b - amount*. The second axiom for the *balance?* method has a slightly different form in the sense that there is no condition and when the transition *balance b* → *balance b* is possible, the *balance?* method synchronizes with the gate *hasBalance b* in order to output the account balance.

It is important to mention that due to the usage of Petri Nets as a way of expressing behavior, the concurrency in a CO-OPN model is managed "for free". In fact, although places in a CO-OPN class can be loosely associated to the traditional notion of class attributes, its semantics is very different. The consumption of a resource in a place means the resource no longer exists – this makes it impossible, for example, for two simultaneous calls to the same method to succeed if they access the same resource.

2.2.3 *Object Coordination Model and Transactional Semantics*

Apart from the underlying semantics of Petri Nets, the CO-OPN language also employs a coordination model that allows expressing that the execution of a given method requires the *simultaneous*, *sequential* or *disjunctive* occurrence of other events in the object network that composes the system. As an example, let us imagine how we would implement a *withdraw _ _ : integer string* method in class *BankingUser* (refer to figure 3). This method including two parameters (*amount* and *account identifier*) has as purpose to delegate the activation of the money withdrawal to method *withdraw* of an Account object with the given account identifier. This can only happen if the user is already logged in. The CO-OPN axiom that describes this behavior is:

```
withdraw am accId With us .  isLogged // acc .  checkAccId accId
                    // acc .  withdraw am::
    accounts acc, userState us -> accounts acc, userState us;
```

In this axiom we can see that the method *withdraw* is synchronized with the occurrence of three other events: the user has to be logged, the account has to have the correct identifier and the *withdraw* method of that account object has to be possible. The "//" operator states that the events must happen simultaneously. A CO-OPN method call can be also synchronized with a simple event or with an event sequence or disjunction – represented respectively by the ".." and "+" operators.

[2]The underscore(s) after the name of the method represents the position of the method parameters in the method call

The execution of any CO-OPN method (synchronized or not) is *transactional* in the sense that it is either fully executed or the model's state does not change.

3. CO-OPN, SUT AND TEST GENERATION

By choosing CO-OPN as a specification language for model-based testing we naturally include into the set of SUTs we can test concurrent, non-deterministic and transactional systems. This section provides a discussion on how SUTs with these features have influenced our choices while designing SATEL and while producing CO-OPN specifications that allow model-based testing.

3.1 Stimulations and Observations

Test cases are in principle sequences of stimulation/ observation pairs over the SUT's interface signature. Given that we are performing model-based testing, we require an isomorphism to exist between the model's interface and the SUT's interface. This is an essential assumption of the approach without which it becomes impossible to use the generated tests. In terms of the CO-OPN model we can map the stimulations of the SUT into synchronizations (including the "//", ".." and "+" operators) of method events and observations of the SUT into synchronizations of gate events. This formalization of stimulations and observations has the advantage of being straightforward. On the other hand we entirely leave the complexity of applying stimulations and calculating the test verdicts up to the test driver machinery.

3.2 Non-Determinism and Test Representation Formalism

Taking into consideration that test cases are execution traces of the SUT, their natural representation is as sequences of events (or stimulation/observation pairs, as we have previously defined). Simple temporal logics such as *Traces* [14] can describe such executions in a model, but are insufficient to discriminate different non-deterministic behaviors. Given that CO-OPN allows non-determinism (we can for example declare two different methods with the same fire condition) we have chosen as test representation formalism the HML (Hennessy Milner Logic) temporal branching logic which includes *and* and *not* operators. The unit event in our HML formulas is the stimulation/observation pair. Leaving the formal equivalence relation issues outside the scope of this paper [15], HML allows us to be precise enough to test accurately non-deterministic aspects of the SUT.

As we have previously mentioned in the paper, we consider both valid and invalid behaviors of the SUT as test cases. In that sense we need to add to the HML formulas a logic value *true* or *false* in order to distinguish valid from invalid behaviors. Our test cases are thus pairs of the form $< f, r >$, where f is an HML formula with stimulation/observation pairs as events, and $r \in \{true, false\}$.

3.3 Transactional issues of CO-OPN

CO-OPN's transactional semantics makes it possible to automatically reject certain operations if the state of the model does not allow executing them. For example, in figure 4, the *withdraw amount: integer* operation (or method) is only successful if the condition $(b >= amount) = true$. In case the state of the model is such that this condition is *false*, the model will simply refuse the execution of the operation. These semantics are interesting for modeling because they allow treating operations in a positive way – an operation is executed if and only if the state of the model allows it, otherwise nothing happens. All error situations and inconsistent states are thus avoided. However, most programming languages do not implement these semantics and real SUTs usually react to (distinct) impossible operations with (distinct) error codes. Methodologically speaking it is thus important to model those impossible situations as observable events of the system. Coming back to the model in figure 4, this means it would be interesting to add a second behavior for the *withdraw amount: integer* method, such as:

```
(b >= amount) = false => withdraw amount // errorLowBalance ::
                    balance b -> balance b;
```

This axiom states that if we try to withdraw an amount of money superior to the balance of the account a gate event *errorLowBalance* would occur. Given this new operation it would be possible to associate a stimulation *withdraw amount* (where *amount* is superior to the current balance of the account) with an observation *errorLowBalance* in order to test this behavior.

4. SYNTAX AND SEMANTICS OF SATEL

SATEL is a language for expressing test intentions for CO-OPN specifications. The language should be precise enough to tackle in depth the model the test engineer wishes to produce tests from, but at the same time generic and simple enough to accomplish it without exposing the complexity of the test generation engine itself. Given that different test intentions can cover different functionalities expressed in the model (we have relaxed the need for pertinence of the test set as formalized in [16]), we have also designed the language in a way that test intentions can be reused. In fact we have defined test intentions as modules that may be composed, giving rise to the possibility of devising in the future a methodology for testing systems in a compositional way, possibly reusing previously defined test intentions. A formal description of the syntax and semantics of SATEL may be found in [17].

4.1 Syntax of SATEL

A *test intention* is written as a set of HML formulas with variables, which in the subsequent text we will call execution patterns. The variables correspond to the three dimensions of a test case, namely:

- the shape of the execution paths;
- the shape of each stimulation/observation pair inside a path;
- the algebraic parameters of the methods or gates inside the stimulation/ observation pairs.

A test intention is thus written as a set of partially instantiated execution patterns, where the variables present in those patterns are by default universally quantified. All the combined instantiations of the variables will produce a (possibly infinite) number of test cases.

By constraining the domains of the variables in an execution pattern, the test engineer is able to produce test cases that accurately reflect his/her intuition behind a test intention. For each kind of domain we have devised a number of functions and predicates that allow modeling test intentions. The functions have as co-domains the *integers* and *booleans*, which are native data types of SATEL. The predicates are the typical binary predicates for *integers* (==,<>,>,<,<=, >=) and for *booleans* (==,<>).

- **Variables over the shape of execution paths**: these variables are constrained by using functions that discriminate the of shape HML formulas. In particular we have implemented the following functions that have HML formulas as domain: *nbEvents* – number of events in an HML formula; *depth* – length of the deepest branch of an HML formula; *sequence* – *true* if the HML formula contains no *and* operators; *positive* – *true* if the HML formula contains no *not* operators; *trace* – *true* if the HML formula contains no *and* or *not* operators.

- **Variables over Stimulations/Observations**: given that stimulations and observations are respectively synchronizations of method and gate events, we have devised a number of functions that discriminate the shape of those synchronizations. In particular we have implemented the following functions that have stimulations or observations as parameters: *simpleEvent* – *true* if the stimulation or observation is composed respectively of only one method or gate call; *onlySimultaneity*, *onlySequence*, *onlyAlternative* – *true* if there are only respectively simultaneity ("//"), sequence ("..") or alternative ("+") operators present in the synchronization *nbSynchronizations* – number of simple events in the synchronization.

- **Variables over algebraic parameters of methods or gates**: given the fact that these variables represent values of algebraically defined sets, we use algebraic equations in order to limit the elements from those sets. If we take the example of figure 2, a possible algebraic constraint would be *((a = newChallenge 1 2 3 4) = true)*, which would limit an algebraic variable called *a* to the only possible value of *(newChallenge 1 2 3 4)*.

4.1.1 Declaring test intention rules

Each test intention may be given by several rules, each rule having the form:

```
[ condition => ] inclusion
```

In the *condition* part of the rule (which is optional) the test engineer is able to express constraints over variables – those mentioned in the above list of items. In the *inclusion* part of the rule the test engineer can express that a given execution pattern is included in a named test intention. Assuming given a CO-OPN specification, consider the following rule (where f is variable over execution paths):

```
nbEvents(f) < 5 => f in SomeIntention
```

This rule would produce all possible test cases for that specification that include a number of events inferior to 5. These test cases would become part of the test set generated by the test intention *SomeIntention*.

On the other hand it is possible to declare multiple rules for the same test intention. Let us add to the previous rule the following one, where *aMethod* and *aGate* are respectively ground stimulations and observations:

```
HML{<aMethod,aGate> T} in SomeIntention
```

The set of test cases produced by *SomeIntention* would now become the one produced by the first rule united with the one produced by the second rule. In fact only one test case is produced by the second rule given that there are no variables in the execution pattern *HML{<someMethod,someGate> T}*.

An interesting feature of the language is that it allows reusing rules by composition, as well as recursion between rules. Consider the following set of rules where f and g are variables over the shape of execution paths:

```
          HML{<aMethod,aGate> T} in AnotherIntention
        HML{<aMethod',aGate'> T} in AnotherIntention
f in AnotherIntention, g in AnotherIntention => f .  g
                    in AnotherIntention
```

These rules would produce an infinite amount of test cases which include sequences of the stimulation/observation pairs *<aMethod,aGate>* and *<aMethod',aGate'>* in any order and in any length. In fact, the third rule for *AnotherIntention* chooses non-deterministically any two test cases generated by any rule of the test intention and builds a new test case based on their concatenation[3].

The composition of test intentions is a very important feature given that it allows establishing a methodology for building test intentions that cover progressively larger parts of the SUT. We are currently investigating these methodological issues and their impact on how to write test intentions. A direction for this research is that, using SATEL, *top down* or *bottom up* construction of test cases is possible.

4.1.2 Variable quantification constraints

All the variables used in a test intention rule are universally quantified, while satisfying the constraints expressed in the condition of that rule. This will produce test sets which are the combination of all the possible values the variables assume in the execution patterns on the right side of the rule. In some cases this is not practical because we may want to select randomly a value from a given domain – this corresponds to uniformity hypothesis as described in [16], whereas the previously presented constrains correspond rather to regularity hypothesis.

In order to deal with this random aspect necessary for test generation, we have included in SATEL two unary predicates that may be seen as quantifiers for variables of our language. These quantifiers may be applied to any variable of the language (shape of execution paths, shape of synchronizations or algebraic parameters of methods). However, they will only quantify directly the variables which represent the algebraic parameters of methods. In what concerns the remaining variable types, they will be quantified indirectly in the sense that the quantification will propagate to all the method parameters included in the execution patterns that those variables stand for.

- ***uniformity(var)***: all method parameters directly or indirectly in the scope of *var* will be attributed one random value;
- ***subuniformity(var)***: all method parameters directly or indirectly in the scope of *var* will be attributed one random value for each of the equivalence classes in the semantics of the method they belong to.

As an example, consider the following rule for the test intention *TestWithdraw* where *amount* is a variable of the ADT *integer* type and g is a variable of the type *observation*:

```
               subuniformity(amount) =>
HML{<deposit(10),null><withdraw(amount),g> T} in
                     TestWithdraw
```

Using an object of type *Account* as specification (see figure 4), this test intention would generate for example the following valid test cases:

`<deposit(10),null`[4]`><withdraw(5),null>`, *true*
`<deposit(10),null><withdraw(15),errorLowBalance>`, *true*

In fact the *subuniformity* predicate allows choosing two values for the *amount* variable, one for each fire condition of the method *withdraw*. We have defined two axioms for *withdraw* with the complementary conditions $(b >= amount) = true$ and $(b >= amount) = false$. Operationally we choose one value satisfying each of those conditions, hence covering the two equivalence classes of *withdraw*.

4.2 Semantics of SATEL

The abstract semantics of SATEL corresponds to three consecutive steps:

1. Expansion of the test patterns defined in the test intentions by instantiating the variables to their possible values. All variables are instantiated except for the ones marked with the *subuniformity* quantifier and the observation variables. All the combinations of all instantiations yield a first batch of partially HML formulas;

2. For all HML formulas generated in step 1, instantiation of the variables marked with the *subuniformity* predicate and the observation variables. The instantiation of these variables provide the oracles for those formulas.

3. Checking of the validity of the HML formulas produced in step 2 w.r.t. the CO-OPN specification.

This abstract view of the semantics is not tractable operationally. In fact, to compute the semantics of SATEL's test intentions we use logic programming (which mixes the three steps). In order to instantiate the remaining variables from step 1, we have built a translator of CO-OPN specifications

[3]The "." is the concatenation function between test patterns.

[4]The *null* keyword corresponds to the absence of observation.

to the logic programming language Prolog [18]. By computing the Prolog translated specification with the partially instantiated HML formulas, we are able to fully instantiate them. In particular, to calculate the variables marked with the *subuniformity* quantifier we make use of the *unfolding* technique explained in [19]. The technique involves marking the Prolog translation of the equations specifying each algebraic operation, at certain choice points. These choice points correspond to goals in Prolog for which the solutions will be values inside the equivalences classes of those operations. For an example of application of the unfolding technique, see [20].

We decide of the validity or invalidity of an HML formula by checking its satisfaction in the CO-OPN specification. The Prolog computation of the specification will only instantiate variables in step 2 that lead valid behaviors according to the specification. However, in step 1 we can already have invalid behaviors given that the test intention rules are defined by the test engineer and may include any sequence of stimulation/observation events. In order to decide about the satisfaction of an HML formula in a specification we proceed in the following way:

- the HML formula is *true* if the stimulation/observation pairs that compose the formula can be either computed in the Prolog translation of the specification or instantiated sequentially through all the branches until the end of the formula;
- the HML formula is *false* if some stimulation/observation pair of the formula is fully instantiated but cannot not be computed in the Prolog translation of the specification. In that case we discard the remaining of the formula after that stimulation/observation pair.

5. CASE STUDY – TESTING THE BANKING SYSTEM

In figure 5 we provide a full example of usage of our test intention language for defining a test intentions module for the Banking system. This example is written in the concrete syntax of the language which we have implemented in a toolset for experimentation.

While writing the test intentions we have assumed that for each stimulation of the Banking CO-OPN model there are two possible observations: the positive one means the operation was successful and starts with prefix *OK*; the negative one starts with prefix *error*.

The module *TestBanking* acts over Class *Banking* (as defined in the *Focus* field) and defines several distinct test intentions declared in the fields *Intentions*[5]. In figure 5 the variable names are presented in bold font in order to make explicit where the test patterns will be expanded. The types for those variables are declared in the *Variables* field.

- **axiom 1** is composed of only one stimulation/ observation pair with two variables, *usr* for the user login and *obs* for the observed output. When instantiating *usr* and *obs* we get all valid and invalid behaviors of the login operation for all existing users. However, the *subuniformity* predicate allows making use of the semantics of the model by performing equivalence analysis and picking only two values for the *usr* variable – one for the case where the user exists and one for when the user doesn't exist. Four test cases are then generated by axiom 1: the user exists and gets logged in (valid test case); the user doesn't exist and the SUT issues an error code (valid test case); the user exists but the SUT issues an error code (invalid test case); the user doesn't exist but the SUT allows him/her to log in (invalid test case).

```
TestIntentions TestBanking Focus Banking;
   Interface
      Intentions
         login;
         insertPasswords;
         withdraw;
         parallelLogin

   Body
      Intentions
         nWrongPins

      Use
         Boolean
         Pin

      Axioms
1        subUniformity(usr) => HML(<loginUser(usr) with obs> T in login;

2        [] in nWrongPins;

2        f in nWrongPins => f . HML(<loginUserPass(newString(mario),newPin(1 1 1 1))
             with errorLoginUserPass> T) in nWrongPins;

3        f in nWrongPins & nbEvents(f) < 4 => HML(<loginUser(newString(mario))
             with OKLoginUser>) . f . HML(<loginUserPassword(newString(mario),
             newPin(1 2 3 4)) with obs> T)in insertPasswords;

4        subUniformity(am) => f . HML(<deposit(newString(mario),1,100) with depositOK>
             <withdraw(newString(mario),1,am)> with obs> T) in withdraw;

5        HML(<loginUser(newString(mario)) // loginUser(newString(lucio)) with obs> T)
             in parallelLogin;

      Variables
         am : natural
         f : HML
         obs : observation;
         usr : string

End TestBanking;
```

Figure 5: Test Intentions for the Banking SUT

- **axioms 2** makes use of recursion in order to build a sequence of erroneous introductions of passwords by user "mario". We assume a user "mario" exists in the SUT and his password is (newPassword 1 2 3 4);
- **axiom 3** uses the previously defined pattern *nWrongPins* in order to build all possible test cases for behaviors of the *loginUserPassword* operation. Note the usage of the *nbEvents* predicate to constrain the number of stimulation/observation pairs in the sequence of erroneous logins;
- **axiom 4** produces test for the behaviors of the *withdraw* operation. Variable *f* of type HML (in the simplest case, an execution trace) is not instantiated, which means the compiler should generate all possible paths in order for the SUT to reach a state where a *deposit* operation can be executed. Afterwards the *subuniformity* predicate over variable *am* will find two values for this variable: one under 100 (the amount of the deposit) and another over 100;
- **axiom 5** produces tests for the simultaneous login of two users.

[5]The test intentions declared inside the *body* section are auxiliaries for building other test intentions and will not directly produce test cases.

6. CONCLUSION

In this paper we have explored CO-OPN as a modeling language for model-based test case generation. We started by analyzing the CO-OPN and discussing its properties. Afterwards, we have presented the language SATEL which is suitable for producing test cases in a semi-automatic fashion. In the paper we have applied SATEL to a hypothetical Banking system and the results point to a balanced level of abstraction of the language. We are currently validating the approach with an industrial partner and should have reach clearer conclusions in a near future. As future work we plan on broadening the scope of SATEL to other modeling formalisms (e.g., Statecharts), as well as investigating and proposing a methodology for test intention design and reuse.

7. REFERENCES

[1] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[2] M.-C. Gaudel G. Bernot and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, 1991.

[3] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.

[4] Bruno Legeard and Fabien Peureux. Generation of functional test sequences from b formal specifications-presentation and industrial case study. In *ASE*, pages 377–381, 2001.

[5] George Din. Ttcn-3. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 465–496. Springer, 2005.

[6] Michael Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with asml. In *FATES*, pages 252–266, 2003.

[7] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In *FASE*, pages 281–294, 2004.

[8] Yves Ledru, Lydie du Bousquet, Pierre Bontron, Olivier Maury, Catherine Oriat, and Marie-Laure Potet. Test purposes: Adapting the notion of specification to testing. In *ASE*, pages 127–134, 2001.

[9] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In G. Agha and F. de Cindio, editors, *Workshop on Object-Oriented Programming and Models of Concurrency'95*, pages 131–145, 1995. Turin.

[10] David Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[11] Object Management Group. Unified Modeling Language version 2.0, August 2005. URL: http://www.omg.org/technology/documents/formal/uml.htm.

[12] Luis Pedro, Levi Lucio, and Didier Buchs. Prototyping Domain Specific Languages with COOPN. In *Rapid Integration of Software Engineering techniques*, volume LNCS 3943. Springer-Verlag, 2006.

[13] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[14] Phillipe Schnoebelen. *Sémantique du parallélisme et logique temporelle, Application au langage FP2*. PhD thesis, Institut National Polytechnique de Grenoble, 1990.

[15] Cecile Peraire. *Formal testing of object-oriented software: from the method to the tool*. PhD thesis, EPFL - Switzerland, 1998.

[16] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, London, UK, 1995. Springer-Verlag.

[17] Levi Lúcio. Syntax and semantics of satel (semi automatic testing language). Technical report, Software Modeling and Verification Group, University of Geneva, 2006. url: http://smv.unige.ch/tiki-index.php?page=temporaryTechReports.

[18] M. Buffo and D. Buchs. Symbolic simulation of coordinated algebraic petri nets using logic programming. University of Geneva – Internal Note.

[19] Bruno Marre. *Une métode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. PhD thesis, Universite de Paris-Sud – Centre D'Orsay, 1991.

[20] Levi Lucio and Marko Samer. Technology of test-case generation. In *Model-Based Testing of Reactive Systems*, pages 323–354, 2004.

Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs

Shay Artzi Michael D. Ernst Adam Kieżun Carlos Pacheco Jeff H. Perkins
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
{artzi,mernst,akiezun,cpacheco,jhp}@csail.mit.edu

Abstract

A test input for an object-oriented program typically consists of a sequence of method calls that use the API defined by the program under test. Generating legal test inputs can be challenging because, for some programs, the set of legal method sequences is much smaller than the set of all possible sequences; without a formal specification of legal sequences, an input generator is bound to produce mostly illegal sequences.

We propose a scalable technique that combines dynamic analysis with random testing to help an input generator create legal test inputs without a formal specification, even for programs in which most sequences are illegal. The technique uses an example execution of the program to infer a model of legal call sequences, and uses the model to guide a random input generator towards legal but behaviorally-diverse sequences.

We have implemented our technique for Java, in a tool called Palulu, and evaluated its effectiveness in creating legal inputs for real programs. Our experimental results indicate that the technique is effective and scalable. Our preliminary evaluation indicates that the technique can quickly generate legal sequences for complex inputs: in a case study, Palulu created legal test inputs in seconds for a set of complex classes, for which it took an expert thirty minutes to generate a single legal input.

1. Introduction

This paper addresses the challenge of automatically generating test inputs for unit testing object-oriented programs [23, 24, 16, 9, 21]. In this context, a test input is typically a sequence of method calls that creates and mutates objects via the public interface defined by the program under test (for example, `List l = new List(); l.add(1); l.add(2)` is a test input for a class that implements a list).

```
TextFileDriver d = new TextFileDriver();
Conn con = d.connect("jdbc:tinySQL",null);
Stmt s1 = con.createStmt();
s1.execute(
  "CREATE TABLE test (name char(25), id int)");
s1.executeUpdate(
  "INSERT INTO test(name, id) VALUES('Bob', 1)");
s1.close();
Stmt s2 = con.createStmt();
s2.execute("DROP TABLE test");
s2.close();
con.close();
```

Figure 1. Example of a manually written client code using the tinySQL database engine. The client creates a driver, connection, and statements, all of which it uses to query the database.

For many programs, most method sequences are illegal; for correct operation, calls must occur in a certain order with specific arguments. Techniques that generate unconstrained sequences of method calls are bound to generate mostly illegal inputs. For example, Figure 1 shows a test input for the tinySQL database server[1]. Before a query can be issued, a driver, a connection, and a statement must be created, and the connection must be initialized with a meaningful string (e.g., `"jdbc:tinySQL"`). As another example, Figure 7 shows a test input for a more complex API.

Model-based testing [10, 14, 20, 6, 12, 19, 13, 18, 7, 15] offers a solution. A model can specify legal method sequences (e.g., `close()` cannot be called before `open()`, or `connect()` must be called with a string that starts with `"jdbc:"`). But as with formal specifications, most programmers are not likely to write models (except perhaps for critical components), and thus non-critical code may not take advantage of model-based input generation techniques.

To overcome the problem of illegal inputs, we developed a technique that combines dynamic analysis and random testing. Our technique creates a model of method sequences from an example execution of the program under test, and uses the model to guide a random test input generator towards the creation of legal method sequences. Because the model's sole purpose is aiding a

[1]http://sourceforge.net/projects/tinysql

random input generator, our model inference technique is different from previous techniques [8, 22, 1, 25] which are designed primarily to create small models for program understanding. Our models must contain information useful for input generation, and must handle complexities inherent in realistic programs (for example, nested method calls) that have not been previously considered. At the same time, our models need not contain any information that is useless in the context of input generation such as methods that do not mutate state.

A random generator uses the model to *guide* its input generation strategy. The emphasis on "guide" is key: to create behaviorally diverse inputs, the input generator may diverge from the model, which means that the generated sequences are similar to, but not identical to, the sequences used to infer the model. Generating such sequences is desirable because it permits our test generation technique to construct new behaviors rather than merely repeating the observed ones. Our technique creates diverse inputs by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters), and (iii) diverging from models by randomly inserting calls to methods not observed during execution. (Some of the generated inputs may be illegal—our technique uses heuristics that discard inputs that appear to be illegal based on the result of their execution [17].)

In this paper, we make the following contributions:

- We present a dynamic model-inference technique that infers call sequence models suitable for test input generation. The technique handles complexities present in real programs such as nested method calls, multiple input parameters, access modifiers, and values of primitives and strings.

- We present a random test-input generation technique that uses the inferred models, as well as feedback obtained from executing the sequences, to guide generation towards legal, non-trivial sequences.

- We present Palulu, a tool that implements both techniques for Java. The input to palulu is a program under test and an example execution. Palulu uses the example execution to infer a model, then uses the model to guide random input generation. Palulu's output is a collection of test inputs for the program under test.

- We evaluate Palulu on a set of real applications with constrained interfaces, showing that the inferred models assist in generating inputs for these programs. Our technique achieves better coverage than purely random test generation.

The remainder of the paper is organized as follows. Section 2 presents the technique. Section 3 describes an experimental evaluation of the technique. Section 4 surveys related work, and Section 5 concludes.

2. Technique

The input to our technique is an example execution of the program under test. The output is a set of test inputs for the program under test. The technique has two steps. First, it infers a model that summarizes the sequences of method calls (and their input arguments) observed during the example execution. Section 2.1 describes model inference. Second, the technique uses the inferred models to guide random input generation. Section 2.2 describes test input generation.

2.1 Model Inference

For each class observed during execution, our technique constructs a model called a *call sequence graph*. Call sequence graphs are rooted, directed, and acyclic. The edges represent method calls and their primitive and string arguments. Each node in the graph represents a collection of object states, each of which may be obtained by executing the method calls along some path from the root to the node. In other words, a node describes the history of calls. Each path starting at the root corresponds to a sequence of calls that operate on a specific object—the first method constructs the object, while the rest of the methods mutate the object (possibly as one of their parameters). Note that when two edges point to the same node, it does not necessarily mean that the underlying state of the program is the same.

For each class, the model inference algorithm constructs a model in two steps. First, it constructs a call sequence graph for each object of the class, observed during execution (Section 2.1.1). Second, it creates the model for the class by merging all call sequence graphs of objects of the class (Section 2.1.2). Thus, the call sequence graph for the class is a summary of call sequence graphs for all instances of the class.

For example, part (b) of Figure 2 shows the call sequence for $s1$, an object of class of `Stmt` in the program of Figure 1. Part (c) of Figure 2 shows the call sequence graph corresponding to the call sequence in part (b). The graph in part (c) indicates, for example, that it is possible to convert state A to state C either by calling `s1.execute()` or by calling `TS.parse(s1, DR)` and then calling `s1.setStmt(SQLStmt)`.

Figure 3 shows merging of call sequence graphs. The left and center parts show the graphs for `s1` and `s2`, while the right part shows the graph that merges the `s1` and `s2` graphs.

2.1.1 Constructing the Call Sequence Graph

A *call sequence* of an object contains all the calls in which the object participated as the receiver or a parameter, with the method nesting information for sub-calls. Figure 2(b) shows a call sequence. A *call sequence graph* of an object is a graph representation of the object's call sequence—each call in the sequence has a corresponding edge between some nodes, and calls nested in the call correspond to additional paths between the same nodes. Edges are annotated with primitive and string arguments of the calls, collected during tracing. (Palulu records method calls, including arguments and return values, and field/array writes in a trace file created during the example execution of the program under test.)

The algorithm for constructing an object's call sequence graph has three steps. First, the algorithm removes state-preserving calls from the call sequence. Second, the algorithm creates a call sequence graph from the call sequence. For nested calls, the algorithm creates alternative paths in the graph. Third, the algorithm removes non-public calls from the graph.

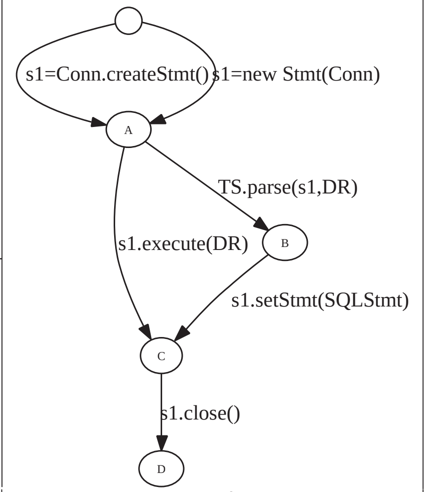
1. Removing state-preserving calls. The algorithm removes from the call sequence all calls that do not modify the state of the (Java) object.

TS≡`tinySQL` Conn≡`Connection`
Stmt≡`Statement`
DR≡`"DROP TABLE test"`
CR≡`"CREATE TABLE test (name char(25), id int)"`
DR≡`"DROP TABLE test"`
IB≡`"INSERT INTO test (name, id) VALUES('Bob', 1)"`

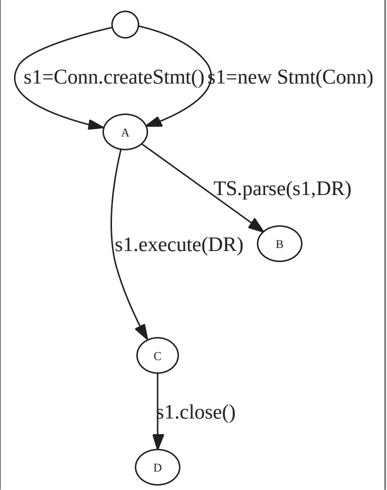
(a) Class and string literal abbreviations

1. s1 = Conn.createStmt()
2. → s1 = new Stmt(Conn)
3. s1.execute(DR)
4. → TS.parse(s1, DR)
5. →→ TSParser.DropTable(s1)
6. →→→ new DropTableStmt(s1)
7. → **s1.setStmt(SQLStmt)**
8. s1.close()

(b) Call sequence for object `s1`



(c) Call sequence graph for `s1`



(d) Public-call sequence graph for `s1`

Figure 2. Constructing a call sequence graph for an object. (a) Abbreviations used in Figures 2 and 3. (b) Call sequence involving object `s1` in the code from Figure 1. Indented lines (marked with arrows) represent nested calls, shaded lines represent state-preserving calls, and lines in bold face represent non-public calls. (c) Call sequence graph for `s1` inferred by the model inference phase; it omits state-preserving calls. The path A-B-C represents two calls (lines 4 and 7) nested in the call in line 3. (d) Public call sequence graph, after removing from (b) an edges corresponding to a non-public call.

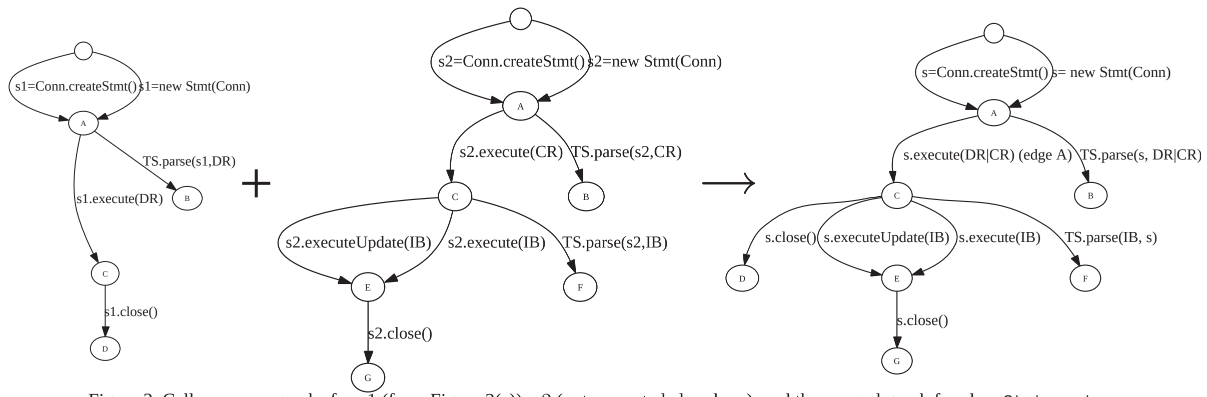


Figure 3. Call sequence graphs for $s1$ (from Figure 2(c)), $s2$ (not presented elsewhere), and the merged graph for class `Statement`.

State-preserving calls are of no use in constructing inputs, and omitting them reduces model size and search space without excluding any object states. Use of a smaller model containing only state-changing calls makes test generation more likely to explore many object states (which is one goal of test generation) and aids in exposing errors. State-preserving calls can, however, be useful as oracles for generated inputs, which is another motivation for identifying them. For example, the call sequence graph construction algorithm ignores the calls in lines 5 and 6 in Figure 2(b).

To discover state-preserving calls, the technique performs a dynamic immutability analysis [3] on the example execution. A method parameter (including the receiver) is considered immutable if no execution of the method changes the state of the object passed to the method as the actual parameter. The "state of the object" is the part of the heap that is reachable from the object by following field references.

2. Constructing call sequence graph. The call sequence graph construction algorithm is recursive and is parameterized by the call sequence, a starting node, and an ending node. The top-level invocation (for the whole history of an object) uses the root as the starting node and a dummy as the ending node[2].

Figure 4 shows a pseudo-code implementation of the algorithm. The algorithm processes the call sequence call by call, while keeping track of the last node it reached. When a call is processed, a new edge and node are created and the newly created node becomes the last node. The algorithm annotates the new edge with primitive and string arguments of the call.

Nested calls are handled by recursive invocations of the construction algorithm and give rise to alternative paths in the call sequence graph. After a call to method c is processed (i.e., an edge between nodes n_1 and n_2 is added to the graph), the algo-

[2]Dummy nodes are not shown in Figures 2 and 3.

```
// Insert sequence cs between nodes start and end.
createCallSequenceGraph(CallSequence cs,
                        Node start, Node end) {
  Node last = start;
  for (Call c : cs.topLevelCalls()) {
    Node next = addNewNode();
    addEdge(c, last, next); // add "last --c--> next"
    CallSequence nestedCalls = cs.getNestedCalls(c);
    createCallSequenceGraph(nestedCalls, next, last);
    last = next;
  }
  replaceNode(last, end); // replace last by end
}
```

Figure 4. The call sequence graph construction algorithm written in Java-like pseudo-code. The algorithm is recursive, creating alternative paths in the graph for nested calls.

rithm creates a path in the graph starting from n_1 and ending in n_2, containing all calls invoked by c.

For example, part (c) of Figure 2 contains two paths from state A to state C. This alternative path containing `TS.parse(s1, DR)` and `s1.setStmt(SQLStmt)` was added because the call to `s1.execute()` (line 3) in part (b) of Figure 2 invokes those two calls (lines 4 and 7).

3. Removing non-public calls. After constructing the object's call sequence graph, the algorithm removes from the graph each edge that corresponds to a non-public method. Thus, each path through the graph represents a sequence of method calls that a client (such as a test case) could make on the class. Ignoring non-public calls in the same way as state-preserving calls would not yield a graph with the desired properties.

For example, in part (c) of Figure 2, the edge corresponding to the non-public method `s1.setStmt(SQLStmt)` gets removed, which results in the graph presented in part (d) of Figure 2.

2.1.2 *Merging Call Sequence Graphs*

After the algorithm creates call sequence graphs for all observed objects of a class, it merges them into the class's model as follows. First, merge their root nodes. Whenever two nodes are merged, merge any pair of outgoing edges (and their target nodes) if (i) the edges record the same method, and (ii) the object appears in the same parameter positions (if the object is the receiver of the first method it must be the receiver of the second, similarly for the parameters); other parameters, including primitives and strings may differ. When two edges are merged, the new edge stores their combined set of primitives and strings.

For example, the call graphs for `s1` and `s2` can be found in left and center parts of Figure 3, while the combined model is on the right. The edges corresponding to `s1.execute(DR)` and `s2.execute(CR)` are merged to create the edge `s.execute(DR|CR)`.

2.2 Generating Test Inputs

The input generator uses the inferred call sequence models to guide generation towards legal sequences. The generator has three arguments: (1) a set of classes for which to generate inputs, (2) call sequence models for a subset of the classes (those for which the user wants test inputs generated using the models), and (3) a time limit. The result of the generation is a set of test inputs for the classes under test.

The input generator works by mixing pure random generation and model-based generation, as we explain below. The generator is incremental: it maintains an (initially empty) *component set* of previously-generated method sequences, and creates new sequences by extending sequences from the component set with new method calls.

Generating test inputs works in two phases, each using a specified fraction of the overall time limit. In the first phase, the generator does not use the models and creates test inputs in a random way. The purpose of this phase is initializing the component set with sequences that can be used during model-based generation. This phase may create sequences that do not follow the models, which allows for creation of more diverse test inputs. In the second phase, the generator uses the models to guide the creation of new test inputs.

An important challenge in our approach is creating tests that differ sufficiently from observed execution. Our technique achieves this goal by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters), and (iii) diverging from models by randomly inserting calls to methods not observed during execution (such sequences are created in the first, random, phase of generation and may be inserted in the second, model-based, phase).

2.2.1 *Phase 1: Random generation*

In this phase, the generator executes the following three steps in a loop, until the time limit expires [17].

1. **Select a method.** Select a method $m(T_0, \ldots, T_K)$ at random from among the public methods declared in the classes under test (T_0 is the type of the receiver). The new sequence will have this method as its last call.

2. **Create a new sequence.** For type T_i of each parameter of method m, attempt to find, in the component set, an argument of type T_i for method m. The argument may be either a primitive value or a sequence s_i that creates a value of type T_i. There are two cases:

   - If T_i is a primitive (or string) type, then select a primitive value at random from a pool of primitive inputs (our implementation seeds the pool with inputs like `0`, `1`, `-1`, `'a'`, `true`, `false`, `""`, etc.).
   - If T_i is a reference type, then use `null` as the argument, or select a random sequence s_i in the component set that creates a value of type T_i, and use that value as the argument. If no such sequence exists, go back to step 1.

   Create a new sequence by concatenating the s_i sequences and appending the call of m (with the chosen parameters) to the end.

3. **Add the sequence to the component set.** Execute the new sequence (our implementation uses reflection to execute sequences). If executing the sequence does not throw an exception, then add the sequence to the component set. Otherwise, discard the sequence. Sequences that throw exceptions are not useful for further input generation. For example, if the one-method input `a = sqrt(-1);` throws an exception because the input argument must be non-negative, then there is no sense in building upon it to create the two-method input `a = sqrt(-1); b = log(a);`.

Example. We illustrate random input generation using the `tinySQL` classes. In this example, the generator creates test inputs for classes `Driver` and `Conn`. In the first iteration, the generator selects the static method `Conn.create(Stmt)`. There are no sequences in the component set that create a value of type `Stmt`, so the generator goes back to step 1. In the second iteration, the generator selects the constructor `Driver()` and creates the sequence `Driver d = new Driver()`. The generator executes the sequence, which throws no exceptions. The generator adds the sequence to the component set. In the third iteration, the generator selects the method `Driver.connect(String)`. This method requires two arguments: the receiver or type `Driver` and the argument of type `String`. For the receiver, the generator uses the sequence `Driver d = new Driver();` from the component set. For the argument, the generator randomly selects `""` from the pool of primitives. The new sequence is `Driver d = new Driver(); d.connect("")`. The generator executes the sequence, which throws an exception (i.e., the string `""` is not valid a valid argument). The generator discards the sequence.

2.2.2 Phase 2: Model-based generation

Model-based generation is similar to random generation, but the generator uses the model to guide the creation of new sequences. We call the sequences that the model-based generator creates *modeled sequences*, which are distinct from the sequences generated by the random generator. The model-based generator keeps two (initially empty) mappings. Once established, the mappings never change for a given modeled sequence. The mo (modeled object) mapping maps each modeled sequence to the object, for which the sequence is being constructed. The cn (current node) mapping maps each modeled sequence to the node in the model that represents the current state of the sequence's mo-mapped object.

Similarly to the random generator from Phase 1 (Section 2.2.1), the model-based generator attempts to create a new sequences by repeatedly extending (modeled) sequences from the component set. The component set is initially populated with the sequences created in the random generation phase. The model-based generator repeatedly performs one of the following two actions (randomly selected), until the time limit expires.

- **Action 1: create a new modeled sequence.** Select a class C and an edge E that is outgoing from the root node in the model of C (select both class and edge at random). Let $m(T_0, \ldots, T_k)$ be the method that edge E represents. Create a new sequence s' that ends with a call to m, in the same manner as random generation (Section 2.2.1)—concatenate sequences from the component set to create the arguments for the call, then append the call to m at the end. Execute s' and add it to the component set if it terminates without throwing an exception. Create the mo mapping for s'—the s' sequence mo-maps to the return value of the call to m (model inference ensures that m does have a return value). Finally, create the initial cn mapping for s'—the s' sequence cn-maps to the target node of the E edge.

- **Action 2: extend an existing modeled sequence.** Select a *modeled* sequence s from the component set and an edge E outgoing from the node $cn(s)$ (i.e., from the node to which s maps by cn). These selections are done at random. Create a new sequence s' by extending s with a call to the method that edge E represents (analogously to Action 1). If a parameter of m is of a primitive or string type, randomly select a value from among those that decorate edge E. Execute s' and add it to the component set if it terminates without throwing an exception. Create the mo mapping for s'—the s' sequence mo-maps to the same value as sequence s. This means that s' models an object of the same type as s. Finally, create the cn mapping for s'—the s' sequence cn-maps to the target node of the E edge.

Example. We use `tinySQL` classes to show an example of how the model-based generator works. The generator in this example uses the model presented in the right-hand side of Figure 3. In the first iteration, the generator selects Action 1, and method `createStmt`. The method requires a receiver, and the generator finds one in the component set populated in the random generation phase (Section 2.2.1). The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the mo and cn mappings.

sequence s	$mo(s)$	$cn(s)$

`Driver d = new Driver();`
`Conn c = d.connect("jdbc:tinySQL");`
`Statement st = c.createStmt();`
	`st`	A

In the second iteration, the generator selects Action 2 and method `execute`. The method requires a string parameter and the model is decorated with two values for this call (denoted by `DR` and `CR` in the right-most graph of Figure 3). The generator randomly selects `CR`. The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the mo and cn mappings.

sequence s	$mo(s)$	$cn(s)$

`Driver d = new Driver();`
`Conn c = d.connect("jdbc:tinySQL");`
`Statement st = c.createStmt();`
`st.execute("CREATE TABLE test name\`
`          char(25), id int)");`
	`st`	C

3. Evaluation

This section presents an empirical evaluation of Palulu's ability to create test inputs. Section 3.1 shows that Palulu yields better coverage than undirected random generation. Section 3.2 illustrates how Palulu can create a test input for a complex data structure.

3.1 Coverage

We compared using our call sequence models to using universal models (that allow any method sequence and any parameters) to guide test input generation in creating inputs for programs that define constrained APIs. Our hypothesis is that tests generated by following the call sequence models will be more effective, since the test generator is able to follow method sequences and use input arguments that emulate those seen in an example input. We measure effectiveness via block and class coverage, since a test suite with greater coverage is generally believed to find more errors. (In the future, we plan to extend our analysis to include an evaluation of error detection.)

Program	tested classes	classes for which technique generated at least one input		block coverage	
		Universal model	Call sequence model	Universal model	Call sequence model
tinySQL	32	19	30	19%	32%
HTMLParser	22	22	22	34%	38%
SAT4J	22	22	22	27%	36%
Eclipse	70	46	46	8.0%	8.5%

Figure 5. Classes for which inputs were successfully created, and coverage achieved, by using following call sequence models and universal models.

3.1.1 *Subject programs*

We used four Java programs each of which contains a few classes with constrained APIs, requiring specific method calls and input arguments to create legal input.

- **tinySQL**[3] (27 kLOC) is a minimal SQL engine. We used the program's test suite as an example input.
- **HTMLParser**[4] (51 kLOC) is real-time parser for HTML. We used our research group's webpage as an example input.
- **SAT4J**[5] (11 kLOC) is a SAT solver. We used a file with a non-satisfiable formula, taken from DIMACS[6], as an example input.
- **Eclipse compiler**[7] (98 kLOC) is the Java compiler supplied with the Eclipse project. We wrote a 10-line program for the compiler to process, as an example input.

3.1.2 *Methodology*

As the set of classes to test, we selected from the program's public non-abstract classes, those classes that were touched during the sample execution. For classes not present in the execution, call sequence models are not created and therefore the input generated by the two techniques will be the same.

The test generation was run in two phases. In the first phase, seeding, it generated components for 20 seconds using universal models for all the classes in the application. In the second phase, test input creation, it generated test inputs for 20 seconds for the classes under test using either the call sequence models or the universal models.

Using the generated tests, we collected block and class coverage information with emma[8].

3.1.3 *Results*

Figure 5 shows the results. The test inputs created by following the call sequence models achieve better coverage than those created by following the universal model.

The class coverage results differ only for tinySQL. For example, without the call sequence models, a valid connection or a properly-initialized database are never constructed, because of the required initialization methods and specific input strings.

The block coverage improvements are modest for Eclipse (6%, representing 8.5/8.0) and HTMLParser (12%). SAT4J shows a 33% improvement, and tinySQL, 68%. We speculate that programs with more constrained interfaces, or in which those interfaces play a more important role, are more amenable to the technique. Future research should investigate these differences further in order to characterize the programs for which the technique works best, or to improve its performance on other programs.

The results are not dependent on the particular time bound chosen. For example, generation using the universal models for 100 seconds achieved less coverage than generation using the call sequence models for 10 seconds.

[3]`http://sourceforge.net/projects/tinysql`
[4]`http://htmlparser.sourceforge.net`
[5]`http://www.sat4j.org`
[6]`ftp://dimacs.rutgers.edu`
[7]`http://www.eclipse.org`
[8]`http://emma.sourceforge.net`

Class	**Description**	**Requires**
`VarInfoName`	Variable name	
`VarInfo`	variable description	`VarInfoName` `PptTopLevel`
`PptSlice2`	Two variables from a program point	`VarInfo` `PptTopLevel` `Invariant`
`PptTopLevel`	Program point	`PptSlice2` `VarInfo`
`LinearBinary`	Linear invariant ($y = ax + b$) over two scalar variables	`PptSlice2`
`BinaryCore`	Helper class	`LinearBinary`

Figure 6. Some of the classes needed to create a valid test input for Daikon's `BinaryCore` class. For each class, the **requires** column contains the types of all valid objects one needs to construct to create an object of that class.

3.2 Constructing a Complex Input

To evaluate the technique's ability to create structurally complex inputs, we applied it to the `BinaryCore` class within Daikon [11], a tool that infers program invariants. `BinaryCore` is a helper class that calculates whether or not the points passed to it form a line. Daikon maintains a complex data structure involving many classes to keep track of the valid invariants at each program point.

An undirected input generation technique (whether random or systematic) would have little chance of generating a valid BinaryCore instance. Some of its constraints are (see Figure 6):

- The constructor to a `BinaryCore` takes an argument of type `Invariant`, which has to be of run-time type `LinearBinary` or `PairwiseLinearBinary`, subclasses of `Invariant`. Daikon contains 299 classes that extend `Invariant`, so the state space of type-compatible but incorrect possibilities is very large.
- To create a legal `LinearBinary`, one must first create a legal `PptTopLevel` and a legal `PptSlice2`. Both of these classes require an array of `VarInfo` objects. The `VarInfo` objects passed to `PptSlice2` must be a subset of those passed to `PptTopLevel`. In addition, the constructor for `PptTopLevel` requires a string in a specific format; in Daikon, this string is read from a line in the input file.
- The constructor to `VarInfo` takes five objects of different types. Similar to `PptTopLevel`, these objects come from constructors that take specially-formatted strings.

Manually-written test input (written by an expert)	Palulu-generated test input
VarInfoName namex = VarInfoName.parse("x");	VarInfoName name1 = VarInfoName.parse("return");
VarInfoName namey = VarInfoName.parse("y");	VarInfoName name2 = VarInfoName.parse("return");
VarInfoName namez = VarInfoName.parse("z");	
ProglangType inttype = ProglangType.parse("int");	ProglangType type1 = ProglangType.parse("int");
ProglangType filereptype = ProglangType.parse("int");	ProglangType type2 = ProglangType.parse("int");
ProglangType reptype = filereptype.fileToRepType();	
VarInfoAux aux = VarInfoAux.parse("");	VarInfoAux aux1 =
	VarInfoAux.parse(" declaringClassPackageName=, ");
	VarInfoAux aux2 =
	VarInfoAux.parse(" declaringClassPackageName=, ");
VarComparability comp =	VarComparability comp1 =
VarComparability.parse(0, "22", inttype);	VarComparability.parse(0, "22", type1);
	VarComparability comp2 =
	VarComparability.parse(0, "22", type2);
VarInfo v1 =	VarInfo v1 =
new VarInfo(namex, inttype, reptype, comp, aux);	new VarInfo(name1, type1, type1, comp1, aux1);
VarInfo v2 =	VarInfo v2 =
new VarInfo(namey, inttype, reptype, comp, aux);	new VarInfo(name2, type2, type2, comp2, aux2);
VarInfo v3 =	
new VarInfo(namez, inttype, reptype, comp, aux);	
VarInfo[] slicevis = new VarInfo[] {v1, v2};	VarInfo[] vs = new VarInfo[] {v1, v2};
VarInfo[] pptvis = new VarInfo[] {v1, v2, v3};	
PptTopLevel ppt =	PptTopLevel ppt1 =
new PptTopLevel("StackAr.StackAr(int):::EXIT33",	new PptTopLevel("StackAr.push(Object):::EXIT", vs);
pptvis);	
PptSlice2 slice = new PptSlice2(ppt, slicevis);	PptSlice slice1 = ppt1.gettempslice(v1, v2);
Invariant proto = LinearBinary.getproto();	Invariant inv1 = LinearBinary.getproto();
Invariant inv = proto.instantiate(slice);	Invariant inv2 = inv1.instantiate(slice1);
BinaryCore core = new BinaryCore(inv);	BinaryCore lbc1 = new BinaryCore(inv2);

Figure 7. The first code listing is a test input written by an expert developer of Daikon. It required about 30 minutes to write. The second listing is a test input generated by the model-based test generator when following the call sequence models created by a sample execution of Daikon. For ease of comparison, we renamed automatically-generated variable names and grouped method calls related to each class (but we preserved any ordering that affects the results).

- None of the parameters involved in creating a `BinaryCore` or any of its helper classes may be `null`.

We used our technique to generate test inputs for `BinaryCore`. To create the model, we used a trace from an example supplied with the Daikon distribution. We gave the input generator a time limit of 10 seconds. During this time, it generated 3 sequences that create `BinaryCore` objects, and about 150 helper sequences.

Figure 7 (left) shows a test input that creates a `BinaryCore` object. This test was written by a Daikon developer, who spent about 30 minutes writing the test input. We are not aware of a simpler way to obtain a `BinaryCore`.

Figure 7 (right) shows one of the three inputs that Palulu generated for `BinaryCore`. For ease of comparison between the inputs generated manually and automatically, we renamed automatically-named variables and reordered method calls when the reordering did not affect the results. Palulu successfully generated all the helper classes involved. Palulu generated some objects in a way slightly different from the manual input; for example, to generate a `Slice`, Palulu used the return value of a method in `PptTopLevel` instead of the class's constructor.

4. Related Work

Palulu combines dynamic call sequence graph inference with test input generation. This section discusses related work in each area in more detail.

4.1 Dynamic Call Sequence Graph Inference

There is a large literature on call sequence graph inference; we discuss some techniques most closely related to our work. Cook and Wolf [8] generate a FSM from a linear trace of atomic, parameter-less events using grammar-inference algorithms [2]. Whaley and Lam [22] combine dynamic analysis of a program run and static analysis of the program's source to infer pairs of methods that cannot be called consecutively. Ammons et al. [1] use machine learning to generate the graph; like our technique, Ammon's is inexact (i.e., the inferred state machine allows more behaviors than those observed in the trace).

In all the above techniques, the intended consumer of the inferred graphs is a person wanting to gain program understanding. Our end goal is generating test inputs for object-oriented APIs; the consumer of our graphs is a mechanical test input generator, and the model is only as good as it is helpful in generating inputs. This fact imposes special requirements that our inference technique addresses. To be useful for real programs, our call sequence graph inference technique must handle program traces that include methods with multiple input parameters, nested calls, private calls, primitive parameters, etc. On the other hand, the size of the graph is less crucial to us. In addition, the models of the above techniques mostly discover rules affecting one object (for instance, opening a connection before using it). In contrast, our model inference discovers rules consisting of many objects and method calls.

Another related project is Terracotta [25], which dynamically infers temporal properties from traces, such as "event E_1 always happens before E_2." Our call sequence graphs encode specific event sequences, but do not generalize the observations. Using inferred temporal properties could provide even more guidance to a test input generator.

After we publicized our algorithm and experimental results [4], Yuan and Xie [26] presented a very similar algorithm that creates per-object state machines, then combines them. However, they do not present any experimental results.

4.2 Generating Test Inputs with a Model

A large body of existing work addresses the problem of generating test inputs from a specification or model; below we survey the most relevant.

Most of the previous work on generating inputs from a specification of legal method sequences [10, 14, 20, 6, 12, 19, 13, 18, 7, 15] expects the user to write the specification by hand, and assumes that all inputs derived from the specification are legal. In addition, many of these techniques are designed primarily for testing reactive systems or single classes such as linked lists, stacks, etc. whose methods typically can take any objects as parameters. This greatly simplifies input generation—there are fewer decisions to make, such as how to create an object to pass as a parameter.

Like Palulu, the Agedis [13] and Jartege [15] tools use random test input generation; Agedis requires the user to write a model as a UML diagram, and Jartege requires the user to provide JML specifications. The tools can generate random inputs based on the models; the user also provides an oracle to determine whether an input is legal, and whether it is fault-revealing. Compared to Palulu, these tools represent a different trade-off in user control versus automation.

Since we use an automatically-generated model and apply our technique to realistic programs, our test input generator must account for any lack of information in the generated model and still be able to generate inputs for data structures. Randomization helps here: whenever the generator faces a decision (typically due to under-specification in the generated model), a random choice is made. As our evaluation shows, the randomized approach leads to legal inputs. Of course, this process can also lead to creation of illegal structures. In future work, we plan to investigate techniques to minimize this problem.

An alternative approach to creating objects is via direct heap manipulation (e.g., Korat [5]). Instead of using the public interface of an object's class, Korat constructs an object by directly setting values of the object's fields (public and private). To ensure that this approach produces legal objects, the user provides a detailed object invariant specifying legal objects. Our approach does not require a manually-written invariant to create test inputs. Instead, it infers a model and uses it to guide the random search towards legal object states.

5. Conclusion

We have presented a technique that automatically generates structurally complex inputs for object oriented programs.

Our technique combines dynamic model inference with randomized, model-based test input generation to create high-quality test suites. The technique is targeted for programs that define constrained APIs for which random generation alone cannot generate useful tests that satisfy the constraints. It guides random generation with a model that summarizes method sequencing and method input constraints seen in an example execution.

We have implemented our technique for Java. Our experimental results show that test suites generated by our tool achieve better coverage than randomly generated ones. Our technique is capable of creating legal tests for data structures that take a human significant effort to test.

6. References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, Jan. 2002.

[2] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, Sept. 1983.

[3] S. Artzi, M. D. Ernst, D. Glasser, and A. Kieżun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT CSAIL, Sept. 18, 2006.

[4] S. Artzi, A. Kieżun, C. Pacheco, and J. Perkins. Automatic generation of unit regression tests. `http://pag.csail.mit.edu/6.883/projects/unit-regression-tests.%pdf`, Dec. 16, 2005.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, July 2002.

[6] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *IEEE TSE*, 10(1):56–109, 2001.

[7] Conformiq. Conformiq test generator. `http://www.conformiq.com/`.

[8] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, July 1998.

[9] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431, May 2005.

[10] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4*, pages 165–177, Oct. 1991.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, pages 112–122, July 2002.

[13] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *ISSTA*, pages 129–132, July 2004.

[14] D. Hoffman and P. Strooper. Classbench: a framework for automated class testing. *Software: Practice and Experience*, 1997.

[15] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, pages 242–256, Sept. 2005.

[16] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, July 2005.

[17] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Sept. 2006.

[18] Reactive Systems, Inc. Reactis. `http://www.reactive-systems.com/`.

[19] J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, 2003.

[20] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *ICSM*, pages 302–310, Sept. 1993.

[21] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, July 2006.

[22] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, July 2002.

[23] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, pages 196–205, Nov. 2004.

[24] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, Apr. 2005.

[25] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *ISSRE*, pages 340–351, Nov. 2004.

[26] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *AST Workshop*, pages 64–70, May 2006.

Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches

Leila Naslavsky
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
lnaslavs@ics.uci.edu

Henry Muccini
Dipartimento di Informatica
University of L'Aquila - Italy
muccini@di.univaq.it

Debra Richardson
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
djr@ics.uci.edu

ABSTRACT

Practitioners regard software testing as the central means for ensuring that a system behaves as expected. Recently, with the widespread adoption of modeling notations for OO systems, academia and industry are looking at model-based testing as a possible way to complement existing testing techniques.

As a result of this effort, many model-based testing approaches have been proposed. However, the suitability of such approaches for industrial projects is unclear, and their level of adoption is still limited. To better understand their suitability for industrial projects, this paper summarizes a survey that studies automated model-based testing approaches from two perspectives: *level of automation* (of each testing activity) and *ability to track information* among test-related artifacts (e.g. from models to code and vice-versa).

1. INTRODUCTION

Practitioners regard software testing as the central means for ensuring that a system behaves as expected. In traditional software development processes, the source code was the only artifact to be used for testing purposes and many code-level testing techniques have been introduced.

The introduction of automated code-based testing techniques has strongly facilitated the introduction of software testing into practice; automation has reduced the amount of effort spent on technical testing activities and also increased the precision of activities, like result evaluation, often performed by humans and thus more error-prone. Many solutions are currently available in the industry [23] and in the academia [7, 35] that automate some testing activities and reduce testers efforts.

However, since source code is produced at the latest step in the software production process, testing activities are left to the end of the software life cycle. In consequence, schedule slippage, time-to-market pressures, and cost-constraints results in neglected testing. Moreover, since code-based testing uses the implementation to derive test cases it cannot be used alone to test the original expectations about the system.

With the recent widespread adoption of model-driven development, source code is no longer the single source for selecting test cases. Testing techniques can be applied all along the development process, by basing test selection on different pre-code artifacts. Testing against original expectations can be done with model-based testing that adopts high-level models as the basis for test derivation.

Many model-based testing approaches have been proposed so far. However, the suitability of such approaches for industrial projects is unclear, and their level of adoption is still limited. Orientation towards "industrial contexts" imposes, in fact, some extra requirements and constraints over a purely academic testing approach. First, it is not reasonable to assume existence of a formal, complete, and consistent model of the software system. It is, instead, reasonable to assume existence of semi-formal models, such as *UML-based models*. Second, testing in industrial projects can be effective only when the testing effort is "affordable": the testing approaches should support creation of test plans sooner, and they should *automate* most of the testing activities. A new challenge specific to model-based testing consists in automating not only the testing phases, but also the transition between phases, while tracking information among test-related artifacts. As discussed in [14], model-based testing requires the ability to "relate the abstract values of the specification to the concrete values of the implementation". Explicit *relationships* need to be devised between specifications and their implementation (sometimes called mapping), or between specifications and test results (sometimes called traces). The first supports generation of concrete test scripts or execution of abstract test scripts, while the second supports coverage analysis based on the specifications.

Based on such considerations, this paper summarizes a survey that studies automated model-based testing approaches from two perspectives: the first perspective evaluates the level of automation, which is done by evaluating the support for each testing activity. The second perspective aims at understanding the kinds *relationships* used by the approaches, and how they manage these relationships.

The paper is organized so to introduce related work on Section 2. Section 3 describes motivations and goals of this paper. Section 4 proposes our evaluation framework while Section 5 applies the framework over many automated model-based testing approaches. Section 6 concludes the paper.

2. RELATED WORK

In their work, Utting et. al. [33] describe model-based testing as the automatic derivation of concrete test cases from abstract formal models, and their execution. They place model-based testing approaches into a seven-dimension orthogonal taxonomy. The dimensions characterize the approaches with regards to the nature of the model used (e.g. what is modeled, notation used), to the nature of the test generation techniques used (e.g. test selection coverage) and to the nature of the test execution (e.g. on-line, or off-line). Our survey differs from Utting et. al. [33] since we consider model-based testing as consisting of other activities such as coverage analysis and regression testing. Additionally, this survey evaluates scenario-based and state machine-based approaches, whereas Utting et. al. evaluates approaches based on a large variety of paradigms. Moreover, this paper looks into approaches with another perspective: how (and if) these approaches use relationships (mapping among models at different levels of abstraction, and traces among artifacts) to support the different testing activities.

In their work, Prasanna et. al. [27] survey test case generation approaches. They classify these approaches into two categories: specification-based approaches and model-based approaches. However, the description of the difference between these categories is not clear from the paper. Their work provides a shallow description of many approaches, some of which were evaluated in depth and from a different perspective in this paper.

Hartman [12] presents a survey on model based test generation tools. He defines a model based test generator as an automated process that receives as input a formal model of the system under test and a set of directives used to guide the tool in the generation process. He distinguishes between test generators and model based input generators. He also distinguishes between test generators and test automation framework, where the automation framework executes the test sequences without human supervision. The objective of Hartman's survey is to place the AGEDIS project in relation to other tools. He groups the tools into academic and commercial and succinctly describes each of them. It is therefore, a good reference to a list of tool supported approaches, but not an in-depth survey on particular approaches.

In his book, Poston [26] provides a comprehensible explanation on specification-based testing and its technical activities. He includes generation, execution, and evaluation, and measurement as the technical activities that define specification-based testing. In our survey, we also include regression testing as part of the testing activities.

In their book chapter [5], Belinfante et. al. provides an in-depth evaluation of test case generation tools. The book, however, is on model-based testing for reactive systems. It concentrates on approaches that support test case generation and not approaches that automate other testing activities. The approaches included are based on models suitable for describing reactive systems, protocols and distributed systems. These models are in their majority described with formal specification languages. Our survey differs from this one because it includes approaches that accept other (less formal) modeling languages and automates other testing activities.

3. GOALS FOR THIS PAPER

Differently from the previous surveys available in the literature, this paper explores model-based automated approaches with two high level objectives (automation and relationships) discussed in the following paragraphs.

Not arguably, ***automation*** is largely responsible for reducing the amount of effort spent on testing activities. Automated tasks are performed with better precision. For instance, test result evaluation, is more error-prone if done by humans then if automated. Reducing the effort spent on any activity makes it more appealing from the practitioners' perception. For that reason, this survey aims at evaluating the level of automation existent in current model-based testing approaches.

The role played by ***relationships*** among artifacts to support automation of testing activities had long been recognized [8, 28]. Relationships can be established with different purposes: generation of test scripts based on models requires a mapping relationship from concepts in the model to concepts in the implementation; similar relationships are used to support execution of abstract test scripts; measurement of coverage achieved by test suites with respect to models requires relationships from models to the generated test suites, and relationships from test suites to test results. The previous relationships are also used to identify test scripts impacted by modifications to models, so they support selective regression testing.

Creation of some relationships can happen in parallel to creation of artifacts (models, code, test scripts). Other relationships are inferred from existing ones by transitivity. Regardless how they are created, relationships are largely explored for supporting testing activities. Due to their recognized importance, this survey aims at understanding the kinds of relationships used by model-based approaches, and how they support relationship management.

3.1 Scope

Scenarios and state machines have emerged as important modeling perspectives. Approaches evaluated in this paper are a representative subset of automated scenario-based and state-based testing approaches.

Scenario-based approaches were selected because scenarios had been adopted as a means to express requirements and specifications. Scenarios are considered a kind of modeling perspective that faithfully describes requirements. Thus, they are the original expectations about the system, against which the system should be tested. Another influencing factor for evaluating scenario-based approaches is the likelihood that stakeholders would find them easier to understand.

State-based approaches were selected because there is an infrastructure in place for automated testing support based on finite state machines and its variations. State-based languages have often a high degree of formalism, which reduces their likelihood of adoption by practitioners. Thus, approaches and tools based on formal specifications of the system (like AutoFocus, tools accepting as input publicly available specification languages such as SDL or LTS, TestComposer and AutoLink, Cooper, TGV and TorX) are outside the scope of this survey and will be not included in the subset of evaluated approaches. Instead, since UML is the industry's de-facto modeling language, UML-based testing approaches based on state-based models (namely AGEDIS, UMLAUT/Simulator, TESTOR) as are integral part of this study.

4. EVALUATION FRAMEWORK

Figure 1 depicts the evaluation framework. It describes model-based testing as comprised of the following activities: test generation, execution, evaluation [26], coverage analysis and regression testing. With some exceptions, each activity receives as input and provides as output artifacts and relationships among artifacts. Artifacts and relationships output from one activity can be input to the next activity (e.g. a concrete test script output by the test generation activity is used as input to the execution and evaluation activity).

Test generation, execution and evaluation are commonly considered as the main activities that comprise the testing automated process [26]. Coverage analysis and regression testing, however, are not always considered as part of the automated testing process. Coverage analysis informs stakeholders (testers, developers, managers) about the extent to which a set of test scripts covered the artifacts. This information helps them make informed decisions regarding the testing process effectiveness. It is created based on results obtained from test execution and evaluation. Selective regression testing potentially reduces the amount of test scripts to be run after a change to any software artifact is in place. Software is often subject to changes. As a result, new faults can be inserted, so the software needs to be re-tested. The information used to select test scripts is based on previously created test scripts, and on results obtained from the test execution, evaluation, and coverage analysis. As explained, the information used for coverage analysis and for regression testing is produced by the other testing activities. Thus, they have the potential of being integrated into one single approach. Therefore, in addition to supporting test generation, test execution and evaluation, a fully automated model-based testing approach is expected to also support coverage analysis and regression testing (both based on models).

To understand the *level of automation*, the framework evaluates if (and to what extent) the approaches automate each activity in figure 1 with questions particular to each activity. It is expected that the evaluated approaches will provide an integrated, practical, and tool-supported solution. Details are discussed in the sections below. In addition, to understand the *ability to track information* among test-related artifacts, the framework evaluates the kinds of relationships used by each activity (represented by 2-way arrows in figure 1), and the support for relationship management. Relationships in the context of the evaluation framework are further discussed in section 4.5. Given their importance for software testing automation, it is expected that relationships will be used to support each activity. Additionally, given the effort required to establish some of those relationships, it is expected that the approaches will provide support for relationship management.

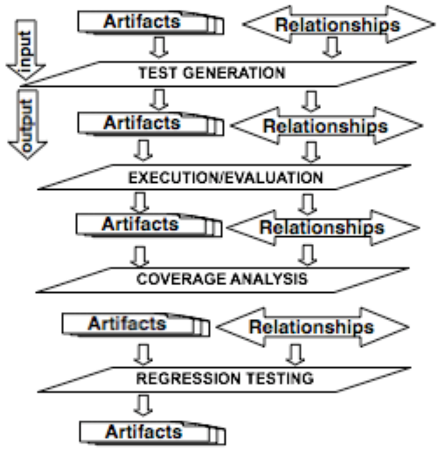


Figure 1 – Model-based Automated Testing

4.1 Test Generation

The test generation activity receives models as input and produces test scripts as output. Test scripts (sometimes named test sequences) are comprised of: (1) set of steps to be followed when testing a program, (2) input and output values. Test scripts are either *abstract* or *concrete*. Abstract test scripts describe the steps a tester should follow when using the system, the inputs to provide and the outputs to expect. In this case, the tester will also evaluate results. Concrete test scripts can be compiled and automatically executed. They consist of calls to methods in the code, the inputs to provide, and the outputs to expect. The evaluation is either made manually by the tester, or automatically by an oracle.

In scenario-based and state machine based approaches, the following items are the possibly input and output by the test generation activity.

Input: scenarios at different levels of abstraction, and/or state-based models (finite state machines and its variations), structural description of the system, code (sometimes named system under test - SUT), other specifications (depending on the approach), relationships (among some items input).

Output: concrete test scripts, provided **relationships** between concepts in model and concepts in the code are available (see relationships in figure 1). Otherwise, abstract test scripts. In addition, it creates **relationships** among artifacts input and artifacts output (e.g. relationship from model input to test script it generated).

Usually, when scenarios are input, they are directly transformed into test scripts (concrete or abstract), which are used to drive the test by exercising the code. If state-based models are input, the tool traverses the model, according to some pre-defined criteria. The criteria are used to guide path selection, as a way of pruning state explosion. These paths are the test scripts generated. Sometimes both scenarios and state-based specifications scenarios are provided as input to the test generation activity. In this case, the scenarios are used to guide the test path selection. Test values in the generated scripts can either be created by the user or generated by the tool using existing techniques such as category partition [22].

To understand the level of automation of the evaluated approaches with respect to test generation, we are interested in learning if the test generation activity produces concrete test scripts. The reason being that concrete test scripts can be automatically compiled and run.

4.2 Test Execution and Evaluation

The test execution and result evaluation receive as input abstract or concrete test scripts output by the test generation activity. Test execution runs test scripts. Result evaluation compares the results obtained against the expected results. Automatic evaluation is done by oracles, which are an element of test scripts. Oracles consist of oracle procedure and oracle information. The procedure uses the information to compare actual to expected behaviors. An oracle, therefore, checks the final result obtained is the expected. It could also check the steps taken to reach a result (or the states through which the system passed to get to that result) are the same steps described in the specification.

The following items are the possible artifacts input and output by the test execution and evaluation activities.

Input: abstract or concrete test scripts, code (sometimes named system under test - SUT), relationships (among some artifacts input).

Output: test results (list of executed and evaluated test scripts with the results obtained: test script passed or failed) and/or test traces (list individual steps executed when the test script was executed). In addition, it creates **relationships** among artifacts input and artifacts output (e.g. relationship from test script input to test result).

If concrete test scripts are input, they are executed and the oracle evaluates the results obtained. The evaluation could compare the results obtained to the results expected, or the behavior obtained to the behavior expected. If abstract test scripts are input, they need to be processed before automated execution can happen. **Relationships** between concepts in abstract test script and concepts in the code are used to support execution. Regardless the input, test results and/or traces are output.

To understand the level of automation of the evaluated approaches with respect to test execution and evaluation, we are interested in learning if models created are used as oracles, and if the check done by the oracle is automatic. The reason is that models describing expected behavior are created anyhow when model-based testing is adopted. They should, therefore, be used as input to leverage automation of execution and evaluation activities.

4.3 Test Coverage Analysis

Test coverage analysis analyzes test results and/or traces to inform stakeholders (testers, developers, managers) about the extent to which a set of test scripts covered the artifacts. This information helps them make informed decisions regarding the testing process. It comes in the form of a report that informs about coverage obtained with respect to different artifacts, or with respect to finer entities that compose the artifacts. Indeed, it depends on the granularity of the test results and test traces available. It also depends on the relationships connecting the test results to other artifacts based on which the coverage is measured. For instance, if relationships are available from individual requirements to test scripts and their results, requirements coverage can be measured.

The following items are the possible artifacts input and output by the test coverage analysis activity.

Input: test results and/or test traces, relationships.

Output: coverage measurement report.

To understand the level of automation of the evaluated approaches with respect to coverage analysis, we are interested in learning if the approach supports it. We are also interested in learning if the analysis is done with regards to artifacts at other levels of abstraction than code.

It's important to note that sometimes coverage analysis, execution and evaluation are performed as if they were one single task. This happens in particular if the coverage is measured with respect to code (statement, method, branch).

4.4 Regression Testing

Software changes require further attention to ensure the quality of the final product will not be affected. When the software is modified, it needs to be retested to reduce the chances that new faults were inserted to the system. Test scripts already run might not need to be re-run. Selective regression testing [11] deals with this issue by reducing the amount of test scripts run after a change is in place. Changes can happen to different software artifacts (code, models), and selective regression testing techniques can be applied for each of them [11, 16]. Ideally, automated model-based testing approaches should support model-based selective regression testing.

The following items are the possible artifacts input and output by the test execution and evaluation activities.

Input: modified scenarios at different levels of abstraction, and/or state-based models (finite state machines and its variations), modified structural description of the system, and the relationships between models, and test scripts.

Output: Subset of selected test scripts to be re-run, and/or subset of the models that should be used as basis for generating the new test scripts.

To understand the level of automation of the evaluated approaches with respect to regression testing, we are interested in learning if the approach supports it. We are, in fact, mostly interested in learning if regression testing is done with regards to artifacts at other levels of abstraction that code.

4.5 Artifacts, Relationships and Relationship Management

Relationships among testing artifacts play an important role on model-based testing automation. Artifacts at their end points characterize them. Changes to these artifacts could imply the relationship does not hold anymore. Thus, to fully achieve automation of model-based testing activities, there is a need to automate management of testing artifacts and the relationships among them (relationship management).

Relationship management is a topic currently addressed by research on software traceability. It aims at supporting (automatically, semi-automatically or manually[1]) creation,

1 This classification was in [30].

persistence, maintenance, and destruction of meaningful relationships across software artifacts [2, 31]. It also aims at describing and classifying those relationships [3, 15, 31].

Software traceability tools automate some software development activities by connecting all inter-relatable software artifacts and supporting relationship management. They exploit the fact that software artifacts have different forms, objectives and semantics; exist at different levels of abstractions and are usually closely inter-related through different kinds of relationships [21, 29]. These relationships can be represented by explicit links, references, and similar name, among other representations [15]. Nevertheless, for those kinds of solutions to be practical, many challenges investigated by the traceability research need to be addressed first. Some challenges exist because stakeholders do not usually maintain (and sometimes not even establish) the relationship representation across the artifacts. As a result, even if the relationships exist, they might become obsolete [9, 31]. Establishing and maintaining relationships among software artifacts is important because relationships can be used in a number of different software engineering activities such as software change impact analysis and software validation, verification and testing [31]. Current traceability approaches that explore validation and verification concentrate on solving inconsistencies among requirements artifacts, design artifacts and code [19, 30]. The use of such approaches to improve automation of model-based software testing is yet to be fully explored.

A commonality between model-based testing and traceability is the need to manage relationships between artifacts. This means traceability infrastructures have the potential to be used with the specific aim of improving testing. This would leverage automation of some model-based testing activities by supporting management of test-related artifacts and their relationships.

4.5.1 *Artifacts and Kinds of Relationships*

To execute each discussed testing activity, the approaches use some kinds of relationships among the artifacts manipulated. For the purpose of this paper, we describe these relationships as *implicit* or *explicit*, *coarse-grained* or *fine-grained*, and *vertical* or *horizontal*. An Implicit relationship is a relationship that the user of the approach is not aware of the existence. It relates two different artifacts (regardless their level of granularity). The user is aware of the existence of the related artifacts. Conversely, an explicit relationship is one that the user is aware of its existence. A Coarse-grained relationship relates artifacts described at a high level of abstraction (e.g. class in class diagram to its implementation). Thus, the level of granularity of the relationship is defined by the level of abstraction of the artifacts at the end-points of the relationship. On the other hand, fine-grained relationships relate artifacts described at a lower level of abstraction. More concretely, we considered a relationship as fine-grained when it related artifacts described at the method level of abstraction or lower levels (e.g. code statement). Vertical relationships relate artifacts described at different levels of abstraction (e.g. relationship between the source and its compiled code), while horizontal relationships relate artifacts described at the same level of abstraction.

Since it is not an objective of this survey to define these relationships formally, figure 2 provides an intuitive explanation and exemplifies these kinds of relationships. For example, R_A exemplifies one instance of a coarse-grained, vertical and explicit relationship. It means that a sequence diagram realizes a use case in a use case diagram. It is coarse-grained because the level of abstraction of a use case is high. It is vertical because use case diagrams are created during the requirements analysis phase, as opposed to the sequence diagrams. It is explicit because in this case the user is aware of its existence (note that it is represented by a thick line). Also, R_D exemplifies one instance of a fine-grained, vertical, implicit relationship. It means a concept in a model (method from a class in a class diagram) is related to (implemented by) a concept in the implementation (method in the code). It is fine-grained because the level of abstraction of a method is low. It is vertical because class diagrams are created during the design phase, and later coded. Also, the code is at a different level of abstraction than the class diagram. It is implicit because in this case the approach uses the relationship, but the user of the approach is not able to explicitly manipulate such relationship (note that it is represented by a dashed line).

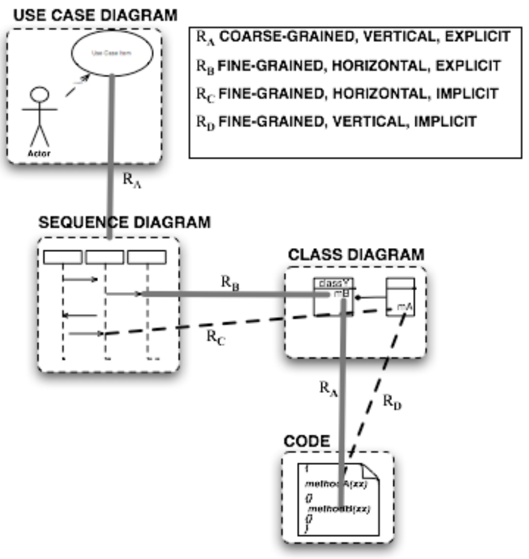

USE CASE DIAGRAM
Use Case Item
Actor
R_A COARSE-GRAINED, VERTICAL, EXPLICIT
R_B FINE-GRAINED, HORIZONTAL, EXPLICIT
R_C FINE-GRAINED, HORIZONTAL, IMPLICIT
R_D FINE-GRAINED, VERTICAL, IMPLICIT
R_A
SEQUENCE DIAGRAM
CLASS DIAGRAM
classY
mB
mA
R_B
R_C
R_A
R_D
CODE
methodA(zz)
methodB(zz)

Figure 2 - Instances of kinds of relationships

5. FRAMEWORK APPLIED TO APPROACHES

We evaluated eleven approaches with respect to the framework described. Six of them are based on scenarios: UCSC-System [18], Sequence Diagram Test Center (SeDiTeC) [10], SCENTOR [34], COWtest pluS UIT Environment (COW_SUITE) [4], Scenario-based Object-Oriented Testing Framework (SOOTF) [32], Testing Object-orienTed systEMs with the unified Modeling language (TOTEM) [6]. Two of them are based on state-based models: Automated Generation and Execution of test suites for DIstributed component-based Software (AGEDIS) [13], and UMLTest [20]. Two of them are based on scenarios **and** state machines: UMLAUT/Simulator [14, 25], and TEst Sequence GeneraTOR (TESTOR) [24]. One of them is based on scenarios **or** state machines: Abstract State Machine Language (AsmL) [1].

This sub-set of model-based testing approaches was selected mainly considering the modeling perspective accepted (scenarios and state machines), the level of formalism of such modeling perspectives, and the number of testing activities supported. This sub-set is mostly representative of academic research, with some exceptions (e.g. AsmL and AGEDIS are approaches developed by industrial research). We are aware of the existence of other tool-supported approaches, particularly commercial tools (e.g. LEIRIOS) that satisfy these criteria, but were not included in the survey. They will be evaluated in future work.

The evaluation was solely based on available publications. The authors created two sets of questions. One set was concerned with the testing activities and aimed at understanding the level of automation of the approach with regards to each activity. The other set was concerned with the kinds of relationships used to support each activity and how these relationships were managed. The process consisted of studying the available publications, answering the questions objectively, and summarizing the findings in the tables shown in sections 5.1 through 5.2. The interested reader is referred to [17], were the questions and answers are detailed.

This section summarizes the main observations about the approaches studied. Each section summarizes information with regards to one testing activity.

5.1 Test Generation

Table 1 summarizes information about the test generation activity of each approach. It has four rows. The first row describes the kinds of artifacts received as input for test generation, the second row describes the artifacts produced as output from the test generation activity, the third row describes the kinds of relationships required by the approach to support test generation (more concretely, relationships required for the creation of the artifacts output), the fourth row describes if the approach support management to these relationships.

5.1.1 Observations

It was expected that scenario-based testing approaches would address the need for using specifications that non-technical stakeholders find easier to manipulate. In particular, this need would be addressed by accepting scenarios at the requirements level. Only few approaches accept such scenarios. They do not, however, address this need. UCSC-System accepts use cases annotated with a formal language for describing pre and pos conditions contracts for the use cases. SOOTF accepts scenarios at the requirements level, and supports the user with a tool to transform this scenario into semi-formal test scenario specifications. AsmL accepts their definition of use cases, which requires the user to learn their language to describe the scenarios programmatically.

With exception of SOOTF, approaches that output concrete test scripts need fine-grained vertical relationships among the artifacts used to generate the test scripts. As already discussed, this kind of relationship is used to map concepts from high-level specifications to low-level ones (code). Since with SOOTF the user is charge of creating the test scenario specification, fine-grained vertical relationships are not used to support test generation.

Table 1 - Test Generation

				UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT		SCENARIOS	REQUIREMENTS	✓				✓						✓
			DESIGN	✓	✓	✓	✓	✓	✓		✓	✓		✓
		STATE MACHINES								✓		✓		✓
		STRUCTURAL DESC.			✓		✓		✓	✓		✓		
		OTHER SPECIFICATIONS		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
		CODE		✓	✓	✓		✓						
OUTPUT		CONCRETE TEST SCRIPTS		✓	✓	✓		✓						
		ABSTRACT TEST SCRIPTS							✓	✓	**	✓	✓	✓
KINDS OF RELATIONSHIPS?	IMPLICIT	COARSE-GRAIN	Vertical	✓	✓				✓					
			Horizontal	✓					✓	✓		✓	✓	
		FINE-GRAIN	Vertical	✓	✓	✓				✓				
			Horizontal		✓							✓	✓	
	EXPLICIT	COARSE-GRAIN	Vertical				✓							
			Horizontal					✓						
		FINE-GRAIN	Vertical				✓							
			Horizontal				✓	✓						
SUPPORT FOR MANAGEMNT OF THESE RELATIONSHIPS?								✓*			X			X

* Manually
** Outputs test requirements

As it could be expected, approaches that use implicit relationships do not support management of such relationships. Inline with this observation, SOOTF uses explicit relationships and provides manual support for relationship management.

5.2 Test Execution and Evaluation

Table 2 summarizes information about the test execution and evaluation activity of each approach. It has six rows. The first row describes the kinds of artifacts received as input for test execution and evaluation activities. Note that apart from the input created by the user (e.g. code), the input received for this activity was output from the previous one. The second row describes the artifacts produced as output from this activity, the third row describes if the approach uses the models created as oracles, and the fourth row describes if the comparison between expected and obtained behavior and results are automated. The last two rows describe the kinds of relationships required by the approach to support test execution and evaluation (more concretely, relationships the approach requires to create the artifacts output), and if the approach support management to these relationships.

Table 2- Test Execution and Evaluation

				UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT		CODE		✓	✓	✓		✓		✓				✓
		CONCRETE TEST SCRIPTS		✓	✓	✓		✓						
		ABSTRACT TEST SCRIPTS								✓				✓
OUTPUT		TEST RESULTS		✓	✓	✓		✓		✓				✓
		TEST TRACES			✓					✓				
ORACLES ARE SPECIFICATIONS?				✓	✓	✓		✓	✓	✓			✓	✓
AUTOMATED EVALUATION?				✓	✓	✓		✓		✓				✓
KINDS OF RELATIONSHIPS?	IMPLICIT	COARSE-GRAIN	Vertical											
			Horizontal											
		FINE-GRAIN	Vertical		✓									
			Horizontal	✓	✓	✓		✓						
	EXPLICIT	COARSE-GRAIN	Vertical											
			Horizontal					✓						
		FINE-GRAIN	Vertical							✓				✓
			Horizontal											
SUPPORT FOR MANAGEMENT OF THESE RELAITONSHIPS?							X	✓*	X	✓*	X	X	X	✓*

* Manually

5.2.1 Observations

It can be observed that approaches whose test generation activity outputs concrete scripts (shown as input in this table) use specifications as oracles and support automated evaluation. It is worth noting, however, only SeDiTeC actually compares the expected behavior to the obtained. The others compare only the results.

Two other approaches compare the expected behavior to the obtained behavior (AGEDIS and AsmL). Note, however, that these approaches receive as input abstract instead of concrete test scripts. For this reason, they require fine-grained and vertical relationships to execute and evaluate such test scripts.

As expected, approaches that use explicit relationships provide support for management of such relationships. However, the support available with these approaches require user to create and manage the relationships. Thus, they are considered as manual support.

5.3 Coverage Analysis

Table 3 summarizes information about the coverage analysis activity of each approach. It has five rows. The first row describes the kinds of artifacts received as input for test execution and evaluation activities. Note that the input received for this activity was output from the previous one. The second row describes the artifacts produced as output from this activity, the third row describes if the approach supports coverage analysis. The last two rows describe the kinds of relationships required by the approach to support coverage analysis, and if the approach support management to these relationships.

Table 3 - Coverage Analysis

				UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT		TEST RESULTS						✓		✓				
		TEST TRACES								✓				
OUTPUT		COVERAGE REPORT						✓		✓				
SUPPORT FOR COVERAGE ANALYSIS?								✓		✓				
KINDS OF RELATIONSHIPS?	IMPLICIT	COARSE-GRAIN	Vertical											
			Horizontal											
		FINE-GRAIN	Vertical											
			Horizontal						✓					
	EXPLICIT	COARSE-GRAIN	Vertical					✓	✓					
			Horizontal											
		FINE-GRAIN	Vertical											
			Horizontal											
SUPPORT FOR MANAGEMENT OF THESE RELATIONSHIPS?								✓*		✓*				

* Automatically creates and persists relationships

5.3.1 Observations

It can be observed that the majority of the approaches do not support coverage analysis. In fact, coverage analysis supported by the AGEDIS tool is based on code coverage (at the method level). Both approaches that support coverage analysis, support automatic creation and persistence of the relationships required for such activity. This can be explained by the nature of the artifacts that stand at the end points of the relationships used to support coverage analysis. These artifacts are test results and test traces. The relationships are created when the artifacts are created by the previous activities (test execution and evaluation). If such relationships were modified, it would result on the modification of the artifacts used as basis for the coverage analysis. As a consequence, the coverage analysis would not consider data that resulted from actual execution of test scripts.

5.4 Regression Testing

Table 4 summarizes information about the regression testing activity of each approach. It has five rows. The first row describes the kinds of artifacts received as input and used as basis for selective regression testing. Note that in this case the artifacts input also include the modified ones (e.g. modified scenarios, state machines, class diagrams, and so forth). The second row describes the artifacts produced as output from this activity, the third row describes if the approach supports regression testing. The last two rows describe the kinds of relationships required by the approach to support regression testing, and if the approach support management to these relationships.

5.4.1 Observations

It can be observed that the majority of the approaches do not support selective regression testing. Those that support rely on the relationships among the artifacts, but do not provide support for automated (or semi-automated) management of such relationships.

Table 4 - Regression Testing

				UCSC-System	SeDiTeC	SCENTOR	COW-SUITE	SOOTF	TOTEM	AGEDIS	UMLTest	UMLAUT	TESTOR	AsmL
INPUT		SCENARIOS	REQUIREMENTS					✓						
			DESIGN					✓	✓					
		STATE MACHINES												
		STRUCTURAL DESC.							✓					
		OTHER SPECIFICATIONS						✓	✓					
		CODE						✓						
OUTPUT		SCENARIOS	REQUIREMENTS					✓						
			DESIGN					✓						
		STATE MACHINES												
		STRUCTURAL DESC.												
		OTHER SPECIFICATIONS						✓						
		CONCRETE TEST SCRIPTS						✓						
		ABSTRACT TEST SCRIPTS							✓					
SUPORT FOR REGRESSION TESTING?								✓	✓					
KINDS OF RELATIONSHIPS?	IMPLICIT	COARSE-GRAIN	Vertical						✓					
			Horizontal											
		FINE-GRAIN	Vertical											
			Horizontal						✓					
	EXPLICIT	COARSE-GRAIN	Vertical											
			Horizontal					✓						
		FINE-GRAIN	Vertical											
			Horizontal					✓						
SUPPORT FOR MANAGEMENT OF THESE RELATIONSHIP?								✓*						

* Manually

6. CONCLUSION

This paper summarized a survey that studies automated model-based testing from two perspectives. We evaluated the level of automation of each approach by evaluating how and if the approach supported each testing activity. We also investigated what kinds of artifacts and relationships are used by the approaches, and how (if) they manage these relationships.

Regarding the first perspective, this paper showed the approaches need better support for some activities. The first is selective regression testing. Support for selective regression testing is important to reduce chances that new faults were inserted to a modified system. It has been implemented in relation to code, specification and architecture. It identifies possibly impacted test scripts based on relationships between the artifact used to generate the test scripts and the test scripts. As observed, the majority of studied approaches do not support selective regression testing. Inline with this observation, it was also observed that the majority of the approaches are still lacking the support for coverage analysis based on high-level artifacts (or based on entities that compose these artifacts).

Not surprisingly, regarding the second perspective, we learned that studied approaches rely on some mechanism to establish relationships between concepts that describe the software artifacts. Relationships can connect concepts across models, or between models and implementation. They are used to support test scripts generation, or to automate the test execution and evaluation.

Support provided for creating and managing relationships varies with the approaches. Some approaches have manual support for managing them, and they require the users to explicitly establish them. When possible, other approaches infer relationships considering name matching and the semantics of the models. For instance, AsmL uses the information input by the user about the namespace to match classes and methods at design and implementation levels. Additionally, approaches that use UML models rely on implicit relationships available from the models. Unfortunately, even when established, these relationships are not maintained. This paper showed model-based testing approaches are currently lacking in support for automated management of the artifacts used and produced, and the relationships among them.

6.1 Recommendation

Given the requirement for automating most of the testing activities, these approaches should improve their support for selective regression testing and coverage analysis.

Regarding selective regression testing, the level of granularity of the end points of the relationships used to support identification of impacted test cases (or impacted models) influences its precision. The more fine-grained these endpoints are, the more precise the selection could be (e.g. code-based regression testing). Therefore, one challenge for model-based selective regression testing is identifying the ideal level of granularity of the artifacts related that would achieve gains with selective regression testing.

Current support for coverage analysis is based on code-level artifacts. Users should be able to obtain other information such as percentage of the requirements, or of scenarios covered by the tests run. This kind of information could be inferred from relationships between the model used to generate the tests scripts, the test scripts and their results. Given the importance of such activity, automated testing approaches should improve their support for coverage analysis based on other (higher level) artifacts than code-level.

Given the requirement for tracking information among related artifacts, these approaches should also improve support for relationship management. In fact, some traceability tools recover relationships between artifacts with different techniques such as dynamic and static reverse engineering [9, 29]. The precision of these is limited, and they require manual intervention to some extent. Nevertheless, integration of traceability tools with model-based testing approaches should improve the level of automation of these approaches. This integration has the potential of leveraging relationship-related testing activities (e.g. regression testing, and coverage analysis). It also has the potential of reducing the burden of relationship maintenance for the users of specification-based testing approaches. Last but not least, it leverages the effort spent on establishing relationships, and thus could encourage the use of traceability tools.

7. ACKNOWLEDGMENTS

The authors acknowledge a 2005 and 2006 IBM faculty award, a research gift from Siemens Corporate Research, and the PLASTIC Project (EU FP6 STREP n.26955)[2], which jointly supported this work.

8. REFERENCES

[1] Asml - Abstract State Machine Language.

[2] Alexander, I., Towards Automatic Traceability in the Industrial Practice, 1st international workshop on traceability in emerging forms of software engineering, Edinburgh, UK, 2002.

[3] Balasubramaniam, R., Matthias,J., Toward Reference Models for Requirements Traceability, IEEE Press, 2001.

[4] Basanieri, F., Bertolino, A., Marchetti, E., The Cow_Suite Approach to Planning and Deriving Test Suites in Uml Projects, Proceedings of the 5th International Conference on The Unified Modeling Language, Springer-Verlag, 2002, pp. 383-397.

[5] Belinfante, A., Frantzen, L., Schallhart, C., Tools for Test Case Generation., in M. e. a. Broy, ed., *Model-Based Testing of Reactive Systems*, Springer LNCS, 2005, pp. 391–438.

[6] Briand, L.C., Labiche, Y., A Uml-Based Approach to System Testing, 4th International Conference on the Unified Modeling Language (UML), Toronto, Canada, 2001, pp. 194-208.

[7] Csallner, C., Smaragdakis, Y., Jcrasher: An Automatic Robustness Tester for Java, 2004, pp. 1025-1050.

[8] Dick, J., Faivre, A., Automating the Generation and Sequencing of Test Cases from Model-Based Specifications, Springer-Verlag, 1993.

[9] Egyed, A., A Scenario-Driven Approach to Trace Dependency Analysis, IEEE Transactions on Software Engineering, 2003.

[10] Fraikin, F., Leonhardt,T., Seditec — Testing Based on Sequence Diagrams, 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261 - 266.

[11] Graves, T. L., Harrold,M. J., Kim, J.-M., Porter,A., Rothermel,G., An Empirical Study of Regression Test Selection Techniques, ACM Transactions on Software Engineering and Methodology, 2001, pp. 184-208.

[12] Hartman, A., *Model Based Test Generation Tools*, 2002.

[13] Hartman, A., Nagin, K., The Agedis Tools for Model Based Testing, 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press, Boston, Massachusetts, USA, 2004, pp. 129-132.

[14] Ledru, Y., du Bousquet, L., Bontron, P., Maury, O., Oriat, C.; Potet, M.-L., Test Purposes: Adapting the Notion of Specification to Testing, International Conference on Automated Software Engineering (ASE), 2001, pp. 127.

[15] Lindval, M., Sandahl, K., Practical Implications of Traceability, Software Practice and Experience, 1996, pp. 1161-1180.

[16] Muccini, H., Dias, M. S., Richardson,D. J., Towards Software Architecture-Based Regression Testing, Journal of Systems and Software, Special Issue on "Architecting Dependable Systems" (2006), pp. 1-7.

[17] Naslavsky, L., Ziv, H., Richardson, D. J., Scenario-Based and State Machine-Based Testing: An Evaluation of Automated Approaches, ISR, University of California, Irvine, 2006.

[18] Nebut, C., Fleurey, F., Traon,Y. L., Jézéquel, J., Automatic Test Generation: A Use Case Driven

[2] Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication, http://www.ist-plastic.org

Approach, IEEE Transactions on Software Engineering, 32 (2006), pp. 140-155.

[19] Nentwich, C., Emmerich, W., Finkelstein, A., Flexible Consistency Checking, ACM Transactions on Software Engineering and Methodology, 2003, pp. 28-63.

[20] Offutt, J., Liu, S., Abdurazik, A., Baldini, A., Ammann P., Generating Test Data from State Based Specifications, The Journal of Software Testing, Verification and Reliability, 13 (2003), pp. 25-53.

[21] Olsen, T., Grundy, J.C., Supporting Traceability and Inconsistency Management between Software Artefacts, IASTED International Conference on Software Engineering and Applications, Boston, MA, 2002.

[22] Ostrand, T. J., Balcer,M. J., The Category-Partition Method for Specifying and Generating Functional Tests, Communications of ACM, 1988, pp. 676-686.

[23] Parasoft, Http://Www.Parasoft.Com/Jsp/Home.Jsp.

[24] Pelliccione, P., Muccini, H., Bucchiarone, A., Facchini, F., Testor: Deriving Test Sequences from Model-Based Specifications, Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005), Lecture Notes in Computer Science, LNCS 3489, St. Louis, Missouri (USA), 2005, pp. 267-282.

[25] Pickin, S., Jard,C., Le Traon,Y., Jéron,T., Jézéquel,J-M., Le Guennec,A., System Test Synthesis from Uml Models of Disrtibuted Software, in D. A. Peled, Vardi, M.Y., ed., Formal Techniques for Networked and Distributed Systems - FORTE 2002, Springer Berlin Heidelberg, Houston, Texas, USA, 2002, pp. 97 - 113.

[26] Poston, R. M., Automating Specification-Based Software Testing, IEEE Computer Society Press, 1996.

[27] Prasanna, M., Sivanandam,S.N., Venkatesan,R., Sundarrajan, R., *A Survey on Automatic Test Case Generation*, Academic Open Internet Journal, 2005.

[28] Richardson, D.J., Aha,S.L., O'Malley,T.O., Specification-Based Test Oracles for Reactive Systems, Proceedings of the 14th international conference on Software engineering, ACM Press, Melbourne, Australia, 1992, pp. 105-118.

[29] Sherba, S. A, Anderson, K. M., Faisal, M., A Framework for Mapping Traceability Relationships, International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03), Montreal, CA, 2003.

[30] Spanoudakis, G., Kim, H., Supporting the Reconciliation of Models of Object Behaviour, Journal of Software and Systems Modelling, 3 (2004), pp. 273-293.

[31] Spanoudakis, G., Zisman, A., Software Traceability: A Roadmap, Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing, 2005.

[32] Tsai, W. T., Saimi, A., YU, L., Paul, R., Scenario-Based Object-Oriented Testing Framework, Third Intermational Conference On Quality Software, 2003, pp. 410 - 417.

[33] Utting, M., Pretschner, A., Legeard, B., A Taxonomy of Model-Based Testing, Department of Computer Science, The University of Waikato, New Zealand, 2006.

[34] Wittevrongel, J., Maurer F., Using Uml to Partially Automate Generation of Scenario-Based Test Drivers, Object-Oriented Information Systems, Springer, 2001.

[35] Xie, T., Marinov,D., Notkin, D., Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests, 19th IEEE International Conference on Automated Software Engineering (ASE'04), 2004, pp. 196-205.