MATAM Advanced Technology Center,
Haifa, 31905
ISRAEL

IBM Research Laboratory in Haifa
Technical Report

A Methodology and Architecture for Automated Software Testing

Ilan Gronau, Alan Hartman, Andrei Kirshin, Kenneth Nagin, Sergey Olvovsky
{gronau,hartman, kirshin,nagin,olvovsky}@il.ibm.com

A Methodology and Architecture for Automated Software Testing

Ilan Gronau, Alan Hartman, Andrei Kirshin, Kenneth Nagin, Sergey Olvovsky

Abstract

In August 1998 the US President's Information Technology Advisory Committee (PITAC) submitted an interim report [15] emphasizing the importance of software to the society. A major theme of the PITAC report was the "fragility" of our software engineering infrastructure. This fragility manifests itself as "unreliability, lack of security, performance lapses, errors, and difficulties in upgrading."

This paper reports on a methodology and software architecture to decrease the fragility of software. Our methodology focuses on the testing activities in the software lifecycle. We advocate a model-based approach to testing, and a philosophy of continuous testing, with design for testing throughout all phases of the software lifecycle from requirements, to specifications, coding, testing, release, and maintenance.

We report on several experiments that applied the methodology with varying degrees of success. We discuss a set of tools, which is constructed according to the architecture for automated testing, and invite contributions to this toolset through the open architecture we advocate. We also report on continuing experiments to refine and improve the methodology.

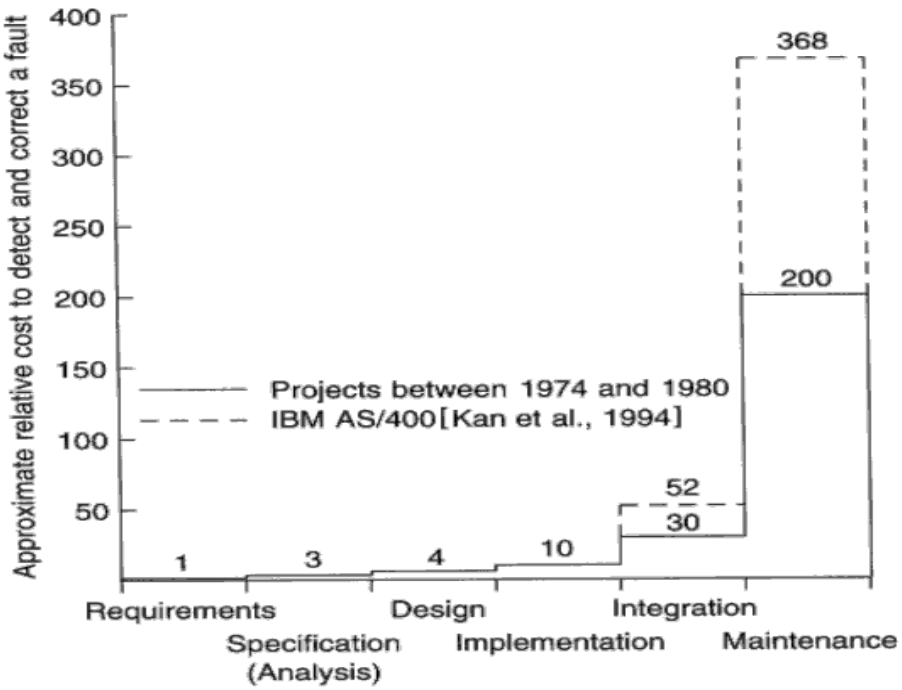
Software Engineering and Testing

In the waterfall model of software engineering, software passes through a sequence of phases that repeat themselves as it proceeds from initial to subsequent releases. These phases are:

- Requirements analysis

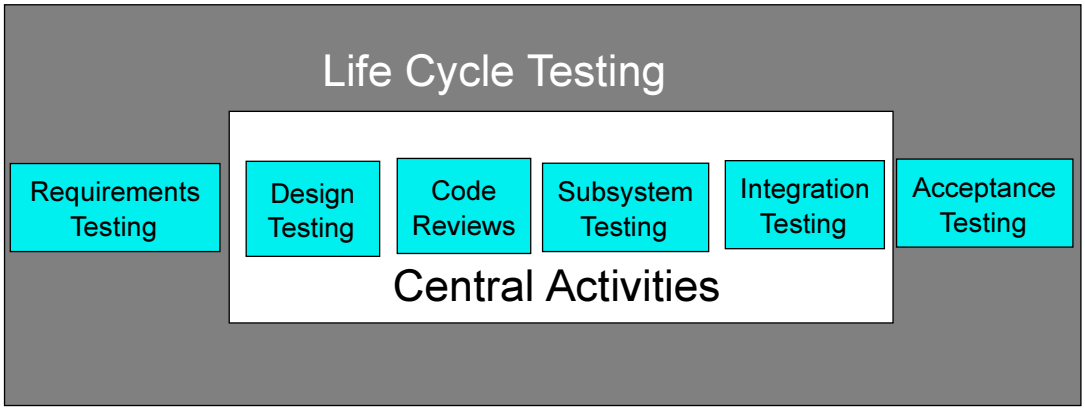
- Design
- Development
- Testing
- Maintenance

This approach to testing – as an activity that follows development – leads to the detection of defects at a late stage in the development process, where the cost of repair is high. Kan et al., [10] report that defects found during maintenance cost more that 300 times as much to repair than the same defect would have cost to repair were it discovered during the requirements analysis.



A more holistic approach to testing is given in Bashir and Goel [1] where they speak of testing activities running concurrent with all phases of development, including requirements testing, design testing, code reviews, subsystems testing, integration testing, and acceptance testing.

Our aim in this paper is to provide a methodology and tools framework for the automation of the central activities in this life cycle model of testing. That is the activities associated with design testing, code reviews, subsystems and integration testing.



Current software development culture partitions the software development lifecycle between four distinct groups of professionals:

- System and requirements analysis: typically people with business process skills and knowledge of the application domain.
- Software design and implementation: software engineers with a detailed knowledge of software methods and practices.
- Software testing and quality assurance: typically people lower in the software development pecking order, with lower levels of education, and fewer high level skills.
- System maintenance: the lowest rank of software professionals, charged with filtering user complaints, and defect reports back to the software development and testing teams.

Economic analysis of software development shows that the major expenditures are in the second and fourth groups, whereas the first and third groups are under capitalized. This lack of analysis and testing creates software products that either never reach production, or contain disastrous defects that are enormously costly in terms of maintenance, loss of business, and in extreme cases, loss of life.

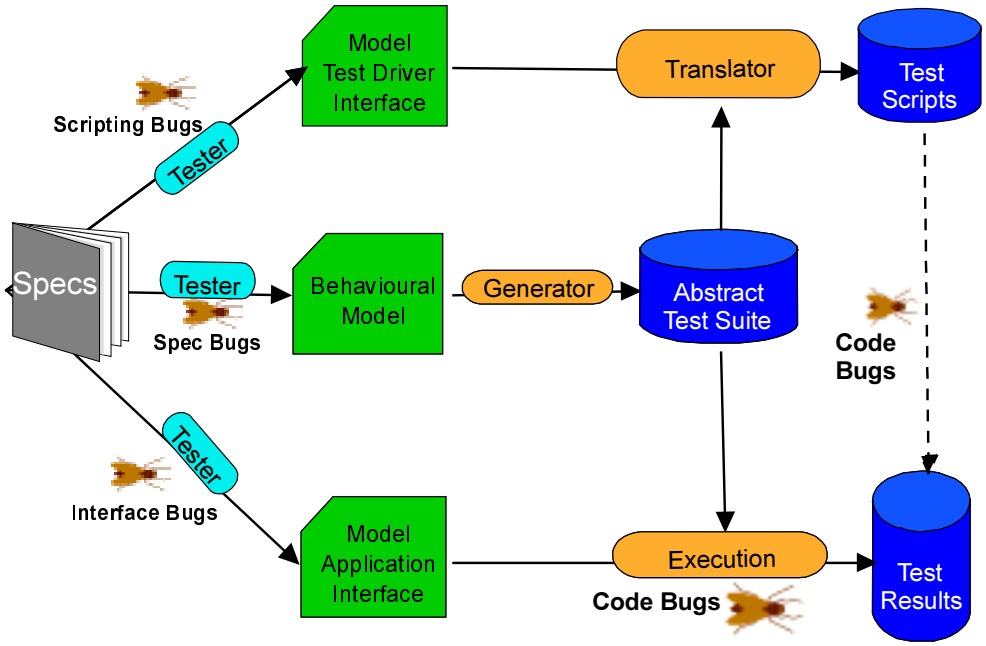
Widmaier, Smidts and Huang [19] report on a controlled experiment in software development where two groups of developers where given identical specifications, and asked to develop a reliable software system using two different development methodologies with the same schedule and budget. A third party, using a methodology similar to that reported in this paper, then tested the resulting systems. Neither of the systems achieved the specified level of reliability, due to insufficient verification efforts based on a systematic automated technology.

Software Modelling

Software modelling is an area of research that has enjoyed great popularity in recent years through the widespread adoption of object-oriented models as an aid to software design [3]. The use of software models for the generation of test suites has also been reported in both an academic setting [4, 8, 12, 13, 14], and in practical experiments [1, 5, 15, 16]. However, the adoption of a specification based modelling strategy for generating test suites has yet to gain much industrial momentum. We will discuss some of the reasons for this in our subsequent section on practical experiments.

We believe that software modelling is an important means of communication between the developers of a system, the software testing professionals, and the user community. The complexities of large software systems can only be accurately communicated by using an abstract language with precisely defined semantics, and mimicking the structures and objects familiar to the users of the system. The formal specifications may not be readily comprehensible to the user community – however formalism is necessary to avoid ambiguities, incompleteness, and inconsistency in communication between members of the development and validation teams.

In the subsequent sections we discuss how modelling can be used practically to give increased efficiency and effectiveness to the software validation process.



The methodology consists of several phases:

1. Write a behavioral model of the software with testing directives. Specification defects are discovered at this stage.

2. Write a testing interface between the model and the application under test. The testing interface may be written in a language understood by an existing test execution engine or by a generic test execution engine. Interface defects are discovered at this stage.

3. Review the model and testing interface with the development team and test team. Further specification defects are discovered at this review.

4. Generate one or more abstract test suites derived from the behavioural model and its coverage criteria using the formal test generator.

5. Translate the abstract test suites to concrete test when the testing interface is for an existing execution engine.

6. Execute the test suites against the software unit under test using an existing test execution engine or using the generic test execution engine. Coding defects are discovered at this phase.

7. Observe the test results and, if necessary, augment or restrict the model or interface and repeat steps 4-5.

Write a Behavioural Model for Test

The first step in the methodology is to write a behavioural model of the software application under test in some formal language. The model is written on the basis of the software specifications, and in conjunction with the code architects and developers. The model also contains testing directives, including descriptions of the coverage goals and test constraints required by the test suite.

In the tools used in our experiments, the behavioural models, coverage directives, and test constraints may all be stored in separate files for ease of configuration management and separation of function. Thus a single behavioural model may be used with different directives and constraints to generate test suites of the same software for different purposes – regression suites, acceptance suites, or full functional testing.

Write a Testing Interface

The second step in the methodology is to create a testing interface between the model and the application under test. The purpose of the interface is to provide the connection between concepts and abstractions used in the behavioural model and those of the software unit and/or the test execution engine.

When testers use an existing test execution engine, the testing interface is prepared by coding an abstract to concrete (A2C) test translation table. The A2C translation algorithm may be written in any programming language (Java, Perl, etc.) to produce concrete test scripts and verification code.

Abstract test suites can also be executed using a generic test driver customized by the testers.

In our experiments, both these options were used. In general an established product with a long development history will already have a customized test driver, and thus it is sufficient simply to translate the abstract tests into the language of the driver – or even to a commercial driver like Mercury's WinRunner[10].

Review Behavioural Model and Testing Interface

A vital part of our process is the review of the behavioural model and testing interface. Testers, architects, and developers of the software conduct the review. The model review discovers inaccuracies, omissions, and contradictions in the specifications. The testing interface review discovers problems related to the interface design and specification. These interface defects are similar to those faced by a customer writing an application or component which interacts with the software unit under test.

In addition to discovering specification and interface defects the review process also discovers defects caused by imperfect communication between

members of the development and test team. Catching these bugs early in the process saves a lot of time and expense later on.

In our experiments, these reviews were important in finding bugs before the cost of their correction became too great.

Generate Abstract Test Suites

The formal test generation tool generates abstract test suites for the behavioural model. These test suites are guaranteed to cover the aspects of the model specified by the tester, while satisfying all the testing constraints imposed.

The generator used in our experiments reports any coverage tasks that cannot be covered by a test satisfying the constraints. Often these indicate bugs in the model or over constrained testing requirements, however, they may also expose defects in the specification.

Execute the Test Suite

The test execution engine can execute an abstract test suite directly against the application under test. This produces a test log that, not only records the test execution, but also compares the outcome of each step in the test with the outcome predicted by the model.

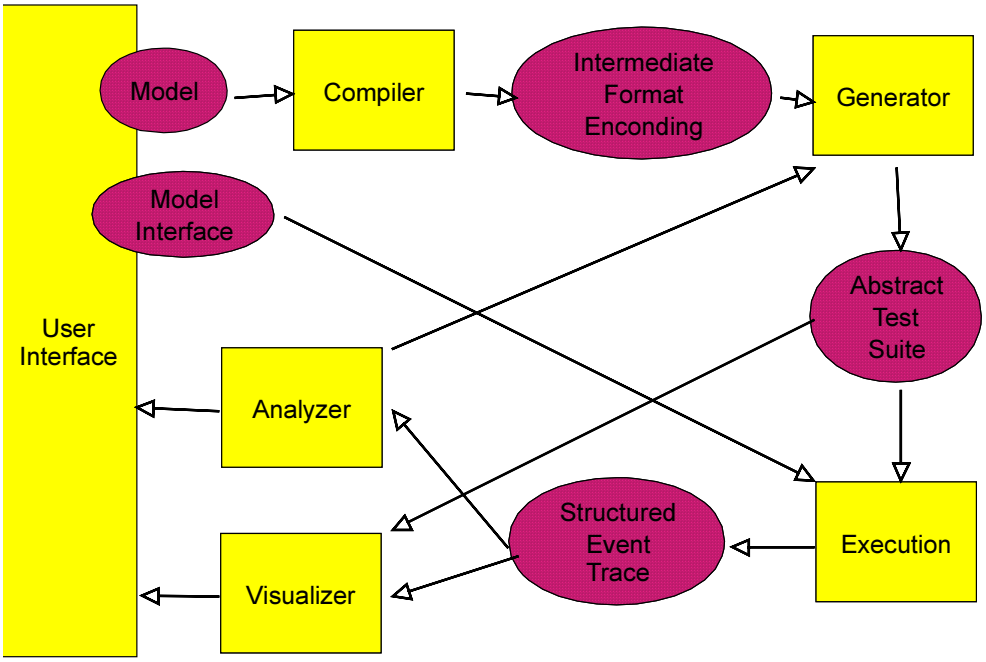
Alternatively, the user's test execution engine stimulates the application under test with stimuli provided from the test suite generated by the behavioural model.

In our experiments, the test execution phase discovered most of the coding and design bugs.

Observe and Iterate

The results of executing the test should be compared with the coverage goals of the test plan. If necessary, further modifications should be made to the model, the testing directives, and/or the test generator's runtime parameters. Further abstract test suites can then be generated to improve the effectiveness of the test.

Our experiments showed that several test suites and several distinct models could be used to test the same piece of software, each one exposing a different set of defects.



Our architecture is designed for open participation and interchangeable components to be developed and implemented as necessary. We recognize that no one modelling language will be adequate to express all software models from all domains. Some groups in the telecommunications industry use SDL to model software, others use the graphical language of the TestMaster tool, and case studies have been reported using Z, CSP, UML, Murphi, SPIN, and others. We maintain that our architectural structure allows for a diversity of modelling languages, and the reuse of existing testing frameworks and execution engines.

We have used a common test generator with two different compilers to generate test suites for models written in languages based on both Objective VHDL[18] and Murphi[6]. We have also demonstrated the feasibility of having two different generators produce test suites in the same abstract format, thus enabling the reuse of the test execution engine on test suites produced by different test generation algorithms. Moreover, we also have examples of a visualization tool and a coverage analyser that utilize the same abstract test format.

The compiler converts the model into the IFE or intermediate form encoding of the model and its testing directives. This is an encoding of a finite state machine (FSM), which describes the behaviour of the system under test, the coverage goals of the test suite, and the restrictions imposed by testing constraints.

This intermediate form – in our experimental tools – was a C++ source file, containing:

1. Classes describing the state variables,
2. Methods for computing the set of all next states from a given state,
3. Methods for generating all start states,
4. Methods for computing the test constraints, and
5. Methods to analyse the states in terms of the coverage directives.

The intermediate form is then compiled and linked together with the test generator code (also in C++) to produce a model-specific test generator in a similar manner to the model-checker produced by Dill et. al.[6].

The test generator then produces an ATS or abstract test suite, which consists of paths through the model satisfying the coverage criteria. This format contains all the information necessary to run the tests and verify the results at each stage of each test case. The user can validate the model by viewing this suite, and use this format to communicate with the developers when a defect is discovered.

Our experimental tools used an XML format for the abstract test suite comprising elements to describe:

1. The set of all state variables and their ranges,
2. The set of all possible inputs to the state machine (stimuli for the software under test),
3. The set of all test cases in the suite, each of which consists of a sequence of transitions.
4. Each transition comprises an input or stimulus, followed by the state entered by the model after responding to the stimulus.

The execution engine reads the abstract test suite and the test interface objects in order to execute the test suite against the application being tested. Each stimulus for each transition is presented to the application under test. The execution engine then queries the state of the application to verify that the application and the model agree on the results of applying the stimulus. The results of the stimulus and verification are written to a structured event trace (SET) in a standard form accessible to all existing and future productivity tools including the analyser and visualiser.

Our experimental execution engine is a set of Java classes, which must be customized by the tester. The tester must code a set of methods:

1. A method for each stimulus to the software under test,

2. A method to query the values of the software artefacts that correspond to the state variables, and

3. A method to instantiate the verification logic, which compares the state predicted by the model with the observed state of the software under test.

The visualiser is capable of showing both the structured event trace and the abstract test suite in a visually informative way to enable the test engineer to comprehend the massive amounts of data generated by automated test generation and execution.

In our experiments, the main visualization tool used was a tree representation of the test suites, with colour codes indicating the success or failure of a particular test suite, test case, or transition. More detailed information on each test element could be displayed by the use of the mouse. Other visualization tools give statistical summaries, and enable the creation of bar charts, histograms, and graphs displaying various aspects of the test suite and its execution trace.

The analyser is capable of reading the structured event trace and identifying areas of the model that may not have been covered sufficiently. It is intended to produce input for the test generator to provide additional test cases. This feedback to the test generator is important in real situations where the translation from abstract tests to actual test runs may not be completely accurate.

In our experiments, this feedback was done manually, by inspecting the test results and making manual changes to the behavioural models, coverage directives and test constraints. In future experiments, we will provide an automated feedback tool.

The user interface includes editors for the models and test interface objects, and a means of activating the tools and viewing their output. This interface has not yet been written, and thus was not used in the experiments.

Experimental Evidence

Several experiments using this methodology and some of the tools, in various stages of their development, have been carried out in various IBM research and development laboratories. Not all of these experiments were an unqualified success, however we believe that the results are sufficiently encouraging to justify continuing refinement of both the tools and methodologies presented in this paper.

Experiment 1: Internet Telephony

The application under test in this experiment was an experimental Internet telephony application. Both the developers and the test team were members of the IBM Research Laboratory in Haifa. Early prototypes of the tools were used, and the methodology was partially implemented. Several other researchers have used this methodology in the telephony software domain [1, 5].

The experiment was judged a success, since the number of defects discovered and their severity was qualitatively higher than expected given the very limited resources used in the test.

Experiment 2: Internet API

Following the in-house experiment, the Haifa Research Laboratory undertook a software-testing contract with a development laboratory in the US. The application under test was the fourth release of an existing product, which is an API for accessing legacy host applications over the Internet.

This experiment was judged by the developers and their management to be an unqualified success. The experiment was completed over a year ago, and no function test escapes have been reported in the past year. Many defects were found both in the new code, and in the existing code base used in the third and prior releases of the product. The developers were so impressed by the quality of our test tools and methodology that they spontaneously gave them the accolade "Function Testing on Steroids" for creating a 'muscle-bound' test suite.

Subsequent to this experiment, and following a competitive evaluation against a competitor's tool, the IBM Software Testing Community Leadership decided to promote wider use of these tools and methodology. Two pilot projects (referred to below as Experiments 3 and 4) were initiated, with the modelling and testing being performed by IBM testers rather than research personnel.

These first attempts at technology transfer required a large investment in the creation of educational materials, documentation, and the setting up of a communications infrastructure connecting the developers of the tools with its users. Both experiments were kick-started with a four-day course in the methodology and technology behind the tools. The course included both lectures and exercises in the art of modelling, the use of the generation tool and the execution tools. A retrospective insight into this course is that the syntax of a modelling language, and the mechanics of tool use can be taught relatively quickly, but that the art of software modelling requires more maturity and experience than can be crammed into a crash course.

Experiment 3: File System Features

This experiment was a retest of certain features of a POSIX compliant file system implemented in the ninth release of an IBM operating system. The records from the previous manually written test of these features were compared with the results of using this automated test generation and execution.

In the original test, eighteen defects were found in the relevant features. The automatically generated test suites found fifteen of these defects and found two new defects that had escaped the earlier test. These results are highly significant since the personnel resources invested in the experiment – including training, model writing, test generation and execution – was 20% less than in the original test using traditional manual test generation.

Moreover, significant changes were implemented in the tools and methodology following this experiment, mainly due to the complexity of the existing test case driver for the system. . The existing test case driver provided methods to invoke file system commands but it did not have the notion of state or the corresponding verification logic. Therefore code test interface mechanisms were required to instantiate the verification logic, which compares the state predicted by the model with the observed state of the software under test.

The management of the testing unit involved were sufficiently impressed by the results of this pilot to commit to future use of the tools in subsequent releases of the product, and to recommend its use to other groups in the development laboratory.

Experiment 4: Internet API Mark 2 – A failure

We initiated a further test of new features of the software product tested in Experiment 2, using an experienced tester to write the models and testing interfaces. This particular experiment did not produce a successful test for a number of reasons. The most obvious factor was the lack of resources for the experiment. Only one person, a very experienced tester, was assigned to the experiment, and at several stages during the experiment this person was required to leave the experiment and contribute to more urgent testing issues in his department. This lack of continuity and under-resourcing led the experiment into a situation where time pressures did not allow for the introduction of a new and unfamiliar testing technique. The experienced tester found efficient ways to test the software using ad hoc methods and custom made scripts without recourse to the novel tools and methodology. Paradoxically, the assignment of an experienced and valuable tester to this experiment was its ultimate downfall.

Experiment 5: Call Centre API

A fifth experiment is currently under way testing a set of APIs for managing call centres. Both the development and testing of this product is split between

the research and development arms of IBM. To date, the experiment is progressing to a successful conclusion with models being written and exploited for test generation in both sites. The defects being found are significant, and the early discovery of several design faults has been a feature of this effort.

Creating the test interface was far from trivial and was very time consuming but it yielded a lot defect discoveries related to the API that ultimately the customer might have encountered. The problem was that the very complexity that made modelling so desirable also meant that the interface was equally complex. The initial development documentation was not sufficient to understand the API usage completely. This meant experimentation was necessary to debug the interface. However, both the model and interface were created before the application to be tested was stable. This is a natural situation when doing unit/function testing. More mature documentation early in the development cycle would simplify the test interface creation and thus enhance parallel test and code development.

The conclusions that we draw from these experiments is that the methodology and tools have a great deal to contribute, but their use must be accompanied by appropriate education, resource allocation, and management involvement in the deployment. Furthermore, the early involvement of testers in the development process improves the quality of the software, but puts additional strain on the documentation process – requiring earlier and more complete documentation than currently available.

These conclusions regarding technology transfer are similar to those reported by Nishiyama, Ikeda, and Niwa[12].

Future Directions

Our focus for the future development of our tools and methodology is on distributed component-based software. We believe that the strengths of our tools lie in the testing of complex interactions between API calls with shared resources and distributed execution environments. To this end we are enriching the modelling language to include the possibility of non-deterministic outcomes for stimuli to the software.

We are also engaged in the development of a new object oriented software modelling language and tools for use in the architecture with a consortium of European universities and industrial partners. This endeavour, called the AGEDIS project includes controlled experimentation with large distributed software being tested at various stages of the development of the tools. An important aspect of the AGEDIS project will be the publication of the open interfaces for IEF, ATS, and SET so that any software testing group can add its own productivity, translation, or generation tools to the suite, without having to duplicate the effort of writing the remaining tools.

References

1. L. Apfelbaum and J. Doyle, Model-based Testing, Proceedings of the 10th International Software Quality Week, QW97, May 1997.

2. I. Bashir and A. L. Goel, Testing Object-oriented Software: Life Cycle Solutions, Springer Verlag New York 1999.

3. G. Booch, Object Oriented Analysis and Design With Applications. Benjamin/Cummings, 2nd edition, 1994.

4. J. Callahan, F. Schneider, and S. Easterbrook, Automated Software Testing Using Model-checking, Proceedings 1996 SPIN Workshop, 1996.

5. J. M. Clarke, Automated Test Generation From a Behavioural Model, Proceedings of the 11th International Software Quality Week, QW98, May 1998.

6. D. L. Dill, A. J. Drexler, A. J. Hu and C. Han Yang, Protocol Verification as a Hardware Design Aid, 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525. http://sprout.stanford.edu/dill/murphi.html

7. A. Hartman and K. Nagin, " TCBeans Software Test Toolkit" in Proceedings of the 12th International Software Quality Week, QW99, May 1999.

8. J. Hartmann, C. Imoberdorf, and M. Meisinger, “UML-Based Integration Testing” in Proceedings of ISSTA 2000.

9. R. M. Hierons, Testing From a Z Specification, The Journal of Software Testing, Verification, and Reliability, 7:19–33, 1997.

10. Kan et. al. Paper quoted in SW Engineering Lecture Notes at http://www.cs.technion.ac.il/~cs234321/LectureSlides/lectureslide.html

11. Mercury Interactive Corporation, WinRunner Test Automation Tool, http://www.merc-int.com/products/winrunner6/.

12. T. Nishiyama, K. Ikeda, and T. Niwa, Technology Transfer Macro-Process A Practical Guide for the Effective Introduction of Technology, International Conference on Software Engineering (ICSE) 2000.

13. J. Offutt and A. Abdurazik, Generating Tests from UML Specifications, Second International Conference on the Unified Modeling Language (UML99), 1999.

14. A. J. Offutt and S. Liu, Generating Test Data from SOFL Specifications, to appear in The Journal of Systems and Software 1999, currently available from http://isse.gmu.edu/faculty/ofut/rsrch/spec.html.

15. A. Paradkar, SALT – An Integrated Environment to Automate Generation of Function Tests for APIs, to appear in Proceedings of ISSRE 2000.

16. R. M. Poston, Automated Testing From Object Models, Aonix White Paper July 1998, http://www.aonix.com.

17. PITAC - Interim Report to the President, National Coordination Office for Computing, Information, and Communications, August 1998, http://www.ccic.gov/ac/interim/

18. M. Radetzki, W. Putzke-Röming, and W. Nebel, A Unified Approach to Object-Oriented VHDL. Journal of Information Science and Engineering 14 (1998), pp. 523-545. http://eis.informatik.uni-oldenburg.de/research/request.shtml

19. J. C. Widmaier, C. Smidts, and X. Huang, Producing More Reliable Software: Mature Software Engineering Process vs. State-of-the-Art Technology, International Conference on Software Engineering (ICSE) 2000.