NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AUTOMATIC TEST CASE GENERATION FOR REACTIVE SOFTWARE SYSTEMS BASED ON ENVIRONMENT MODELS

by

James A. Imanian

June 2005

Thesis Co-Advisors: Mikhail Auguston
James B. Michael

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Automatic Test Case Generation for Reactive Software Systems Based on Environment Models			**5. FUNDING NUMBERS**
6. AUTHOR(S) James Imanian			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The goal of software testing is to expose as many faults as possible. Often one can increase the number of faults detected by running large amounts of test cases, therefore the ability to automatically generate applicable test cases for a System Under Test (SUT), would be a valuable tool. In this thesis an attributed event grammar is designed and used to build a model that describes the environment a SUT must operate in. This event grammar captures events, their precedence or inclusion relation to other events, and attributes of the events. An event is defined as an observable action that has a distinct beginning and end. The high level environment model is then used by a test generator to produce an event trace from which input for the SUT is extracted. Thousands of event traces can be generated. For reactive systems the event trace will have the appropriate time delays between inputs. The feasibility of this approach is proven by implementing a prototype of an automated test generator based on environment models.

14. SUBJECT TERMS Automatic test case generation, attributed event grammar, event trace, event grammar, software testing, testing automation, test generation, real-time systems			**15. NUMBER OF PAGES** 75
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AUTOMATED TEST CASE GENERATION FOR REACTIVE SOFTWARE SYSTEMS BASED ON ENVIRONMENT MODELS

James A. Imanian
Commander, United States Navy
B.S., US Naval Academy, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2005

Author: James A. Imanian

Approved by: Mikhail Auguston
Co-Advisor

James B. Michael
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The goal of software testing is to expose as many faults as possible. Often one can increase the number of faults detected by running large amounts of test cases, therefore the ability to automatically generate applicable test cases for a System Under Test (SUT), would be a valuable tool. In this thesis an attributed event grammar is designed and used to build a model that describes the environment a SUT must operate in. This event grammar captures events, their precedence or inclusion relation to other events, and attributes of the events. An event is defined as an observable action that has a distinct beginning and end. The high level environment model is then used by a test generator to produce an event trace from which input for the SUT is extracted. Thousands of event traces can be generated. For reactive systems the event trace will have the appropriate time delays between inputs. The feasibility of this approach is proven by implementing a prototype of an automated test generator based on environment models.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AEG	Attributed Event Grammar
SUT	System Under Test
TFG	Testing Flow Graph
UML	Unified Modeling Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Mikhail Auguston and Professor Bret Michael for both serving as my thesis advisors and mentoring me as I entered the worlds of computer science and software engineering. They have both made my graduate education experience truly rewarding. I am eternally grateful for the guidance and encouragement they gave me.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The goal of software testing is to expose as many faults or bugs as possible. Automated software testing methodologies can increase the number of faults detected by producing and running large amounts of test cases. The ability to automatically generate applicable, as opposed to purely random, software test cases for a System Under Test (SUT), would be a valuable tool to have when automating this part of the software test cycle. In this thesis an attributed event grammar is designed and then used to build an environment model that describes the environment a SUT must operate in. This event grammar captures events that occur in the desired environment. An event is defined as an observable action that has a distinct beginning and end and has one or more attributes, such as a type, or timing attributes. Events may have a precedence or inclusion relation to other events.

The high level environment model is then parsed and placed in the appropriate form for input to a test generator. The test generator takes this input from the parser, produces an event trace and extracts input for the SUT. Thousands of event traces can be generated. For reactive systems, the event trace will have the appropriate time delays between inputs.

The feasibility of this approach is proven by implementing a prototype of an automated test generator based on environment models. The generator is able to take a parser's structured form of an environment model and generate an event trace. This event trace is then traversed by the generator and the requisite events are collected as actions to be sent to the SUT. The generated sequence of actions provides an interface with the SUT.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION TO THE PROBLEM OF AUTOMATED TEST CASE GENERATION

A. GENERAL SOFTWARE TESTING CHALLENGES

Testing is a resource intensive process that can only achieve subjective goals. Current testing techniques can not guarantee the total absence of bugs, or the software's reliability under all conditions. Testing is used to determine the presence of bugs or faults and to determine if the System Under Test (SUT) implements the required capabilities. A successful test only proves that the SUT was able to handle a certain set of inputs with the SUT in a particular state. Exhaustive testing is impossible to accomplish on non-trivial systems, and even simple implementations of algorithms may hide faults. The most complex problem is testing reactive systems which must take inputs from the environment and produce outputs in real time.

Testing may be done on sub-systems. It is important to note that these sub-systems may be systems in their own right but the fact they passed system level tests does not mean when they are joined with other systems the whole application will function according to requirements and specifications without fault. In "Testing Object-Oriented Systems" Binder points out, system scope failures can result from omissions and from interactions that cannot be produced until all components are exercised in the target environment[1]. Therefore it is important to derive a test plan that is systematic in its ability to comprehensively as possible test the SUT's ability to meet requirements at an application level scope. A solution proposed in this thesis is to use an environment model approach to build these important application level tests.

B. CHALLENGES IN TESTING REACTIVE AND REAL TIME SYSTEMS

Reactive systems are systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination.[2] Two examples of reactive systems are aircraft flight control systems, where the pilots inputs must be translated into mechanical input to flight control surfaces, or the control software

[1] Binder, Robert, *Testing Object-Oriented Systems* (Boston, MA: Addison-Wesley, 2000), 717.

[2] Hhttp://www.cs.nyu.edu/courses/fall02/G22.3033-004/H Accessed June 9, 2005.

used in a nuclear reactor, where the software must take in temperature and other sensor inputs from the core and produce appropriate warnings when conditions are not nominal.

Real time systems are able to respond to inputs from the real world within a predetermined amount of time. Real-time software is characterized by time constraints, that is, time constraints of such a nature that, if a constraint is not met, information is lost.[3] When producing test cases for reactive systems, one must produce a large amount of test data to simulate the prolonged period the SUT must operate. With a SUT that is both reactive and real time, this test data must be fed to the SUT with the appropriate time interval between inputs.

C. SOME REASONS TO AUTOMATE THE TESTING PROCESS

Testing that requires long series of inputs or that has complex relationships, such as timing constraints, are ideal candidates for automation. Additionally the need to generate large amounts of appropriate test data, run that data through a test harness in a mechanized fashion, and to automatically determine if each individual run resulted in a pass or no pass are some of the many other areas of software testing that can benefit from automation. Automated test suites can be rerun to support regression testing or other forms of testing that require the ability to compare baseline results.[4]

Designing the tests and the test data is the most time-consuming portion of the testing process.[5]

D. ADVANTAGES OF AUTOMATED TESTING

Binder lists ten significant advantages of automated testing including:[6]

- Quick and efficient verification of bug fixes
- Decreased cost over manual methods after two or three development cycles.

---

[3] Stephen Schach, *Object-Oriented and Classical Software Engineering*. (Boston, MA: McGraw Hill, 2002),148.

[4] Binder, 803.

[5] Daniel Mosley and Bruce Posey., Just Enough Software Test Automation. (Upper Saddle River, NJ: Prentice Hall PTR, 2002), 10.

[6] Binder, 802.

- Removes errors that occur during manual input.
- Automated comparison is the only repeatable and efficient way to evaluate a large quantity of output.

E. THE MAIN PROBLEMS IN TESTING AUTOMATION

Testing Automation does not come without cost or any disadvantages. Depending on what part of the testing process is automated the following are a few of the difficulties that must be over come:

- How is the appropriate input data going to be generated?
- How are the test cases going to be run on the SUT?
- How are test results going to be verified?

As with any software what type of maintenance will be required on the test cases, test suites or test harnesses that are generated by the automation effort will also be of concern when automating the various phases of software testing.

F. CURRENT AUTOMATED SOFTWARE TESTING METHODS

There are many types of automated testing methods and models that have been developed. The following is a small sample of the current approaches in automated software testing methods. A short description of the principles used by the method, as well as how these methods tackle the main problems in testing automation is discussed.

1. Automated Test Scripts

Automated test scripts can come in two different forms, recorded or programmed test scripts. Programmed test scripts are the product of a programmer writing a piece of software that produces outputs to be feed as input to a SUT. Recorded test scripts are the product of a "recording" of a user inputting data to a SUT. One strength of these automated test scripts have is the fact that if well designed, they are able to stress known or expected application weaknesses.[7] However there are weaknesses as well. With recorded test scripts, rule-based violations can occur at many levels. Recorded test scripts have limitations including hard-coded data, and frequently must be edited before they work properly. Many variations of the same test script must be recorded to cover

[7] Mosley and Posey, 34.

needed test cases. A programmed test script may have all the faults of any other piece of software. The script itself must be verified, that it is conducting the tests you want and validated, that it is conducting the proper tests. Both types of automated scripts result in a large maintenance requirement.

Automated Test scripts therefore address two of the three main problems in testing automation. They create input data by either generating it through another program or by recording actual user inputs to the system. These test cases are then feed as inputs to a SUT either as a flat file, function calls, or via a software wrapper. This method does not solve the problem of how to verify the correctness of the outputs. These automated test scripts can be developed strictly from specifications and therefore can be used in black box testing situations.

2. Universal Modeling Language (UML)

To model the dynamic aspects of a system, UML statechart diagrams can be used. A statechart diagram consists of states, transitions, events and actions and shows a state machine emphasizing the flow of control from state to state.[8] Statecharts are finite state machines extended with hierarchy and orthogonality. Kansomkeat and Rivepiboon present a technique that can automatically generate and select test cases from UML statechart diagrams.[9] This technique transforms UML statechart diagrams into intermediate diagrams, called Testing Flow Graph (TFG). The TFG is then used to generate test cases. These test cases are sequences of function calls that are used as inputs to the SUT. Here again is a method to create input data and run the resulting test cases, but there is no automated method of verifying the correctness of outputs. The use of UML statecharts limits this method to white box testing situations.

3. Automated Testing Based on Java Predicates

Boyapati, Khurshid and Marinov present "Korat", a framework for automated testing of Java programs.[10] Given a formal specification for a method, Korat uses the method precondition(s) to automatically generate all non-isomorphic test cases up to a

---

[8] Grady Booch and others., *The UML Users Guide*. (Reading, MA: Addison-Wesley, 1999).

[9] Supaporn Kansomkeat,and Wanchai Rivepibon, "Automated-Generating Test Case Using UML Statechart Diagrams," in *Proceedings of SAICSIT 2003*, 296.

[10] Chandrasekhar Boyapati and others., "Korat: Automated Testing Based on Java Predicates," in *ACM ISSTA*, July 2002, 123.

given bound on the input. Korat then executes the method on each test case, and uses the method post-condition as a test oracle to check the correctness of each output. With this approach all three main testing automation problems are solved. There are some strict restrictions on when Korat can be used. Programs must be written in Java with the correct pre and post conditions coded for all methods. With this restriction Korat is useful only in white box testing situations.

THIS PAGE INTENTIONALLY LEFT BLANK

II THE ENVIRONMENT MODEL

A. OBJECTIVE OF RESEARCH

The objective of this research was to specify the syntax and semantics of an event based grammar that can model the environment in which a SUT operates. Using this grammar, examples of environment models were to be developed. A prototype implementation of a test driver generator based on these environment models has been developed.

B. THE ENVIRONMENT MODEL APPROACH

The goal of any software testing is to expose as many faults or bugs as possible. Modern automated software testing methodologies increase the number of faults detected through allowing more extensive testing. Current software testing methods allow for automation in the running of test cases, and the monitoring of test case results, but there are relatively few specialized methodologies that actually automate the generation of test cases themselves. There are no uniform methodologies that can be applied over a large range of software.

This thesis suggests a testing approach based on using models of the environment in which the SUT operates. These models would be built using event grammars. There are several advantages inherent in this approach. A wide range of software can be tested using test cases derived from environment models. The model and the tests it would generate can be derived directly from requirements and specifications, even before the code for the system was completed. Lastly this approach is one that can automatically generate a large number of test cases.

C. ENVIRONMENT MODEL STRUCTURE

The event grammar captures events, their sequence or inclusion relation to other events, and the attributes of these events. An event is defined as an observable action that has a distinct beginning and end. The event also has certain attributes associated with it, for example, a missile launch event could have attributes including a time of launch and type of missile.

Events themselves can have two basic relations between them. If the events are dependent on each other they are related by either precedence or inclusion. With precedence Event B is preceded by Event A. With inclusion Event A has one or more instances of Event B that must be concluded before Event A concludes. Two events may have not relation at all or no dependencies between them. These can be called parallel events. Figure 1 depicts a model where events B and C are included in event A, event B precedes event C and events A and D are parallel events

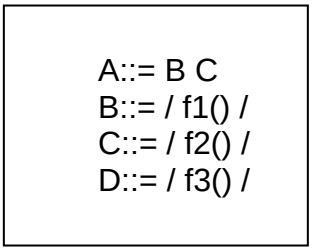

Figure 1. Sample Model Structure.

D. ENVIRONMENT MODEL ADVANTAGES

1. Ability to Generate Valid Data

The ability to automatically generate thousands of test cases does not directly lead to an effective or productive test plan. The true measure of any testing technique is its capacity to get "good" test data. A professional and effective automation effort should create a set of planned tests that corresponds to a set of test requirements that in turn are reflected in the automated tests. Automated tests are effective only when the test data are designed well.[11] The environment model approach has several strengths that allow a test team to create "good" or valid test data.

2. Model Easily Understood

The ability of non-technical personnel to help in the creation or correction of the environment model is a key feature of the environment approach. The model is written at such a high level non-technical personnel should be able to point out both logic errors that could lead to inconsistencies, as well as the absence of certain business rules, or the inclusion of business rules that are not represented in the model.

The following is a portion of the calculator model built for the calculator program tested in this thesis:

[11] Mosley and Posey, XVI.

perform_binary_calculation::= press_binary_op_button enter_number enter_number;

It would not be too difficult for a non-technical person, working on a test team that was testing a calculator that used reverse polish notation, to observe and point out that this perform_binary_calculation is not properly represented in the model. This same person could propose the correct solution of :

perform_binary_calculation::= enter_number enter_number press_binary_op_button;

3. Model Easily Derived from Specification or Use Cases

An environment model can easily be derived from specifications or use cases when available. This is another way to ensure valid data is generated in the test cases. It is possible for the environment model approach to generate a larger set of valid test data than a UML use case approach. A use case describes the functionality of the product to be constructed.[12] Each use case describes a sequence of actions that a systems performs to achieve an observable result of value to an actor.[13] The use case approach uses expected input from the user (environment) to trigger a desired reaction from a particular function or set of functions in the system. The environment model approach models the environment and does not attempt to create "expected" inputs, hence an environment model may produce "unexpected" but still valid test cases for use on the SUT.

4. Model Can Be Used in Verification and Validation

The environment model approach can be used as a tool in both verification and validation of a SUT. The definition of verification and validation in this instance comes from Boehm:[14]

Verification: Are we building the product right?

Validation: Are we building the right product?

An environment model contributes to the verification effort by providing test cases, based on specifications that will test the SUT's ability to meet the specifications. With validation, testers are trying to determine if the SUT is providing the utility needed by a particular set of users. The environment model is not derived from the functionality

[12] Schach, 369.

[13] Simon Bennett and others., UML. (New York: McGraw-Hill, 2001), 27.

[14] Boehm, Barry., "Verifying and Validating Software Requirements and Design Specifications," In *IEEE Software 1 January 1984*, 75-88.

of the SUT but instead from specifications on how the SUT should interact with its environment, i.e. it is a black box testing approach. As such, the tests cases derived from an environment model may highlight deficiencies on how the SUT interacts with its environment. For example a nurse scheduling system for a large hospital may be designed for one user to enter data from a single host. Using non environment model approaches, the scheduling system may pass all tests, the database is large enough, has the required fields, enter data does not produce any race conditions, etc. However test cases from an environment model approach may show that one user can not handle the amount of schedule requests generated by the whole hospital. Therefore the scheduling system in its current form is not the right system for the hospital.

5. Defies Anti-extensionality Axiom

The anti-extensionality axiom states that a test suite that covers one implementation of a specification does not necessarily cover a different implementation.[15] The environment model test cases *will* cover different implementations since they must operate in the same environment. The software wrapper used to provide input to the SUT may vary among implementations but the test cases themselves will not.

6. Model Forms Part of an Ideal Test Suite

A fully automated test process that derives the environment model and test cases from design specifications or source code would be ideal. Automation would eliminate the errors introduced by human involvement in the process. Another benefit would be the reduction in the amount of time required for testing a SUT.

E. A SOLUTION FOR REAL TIME AND REACTIVE SYSTEMS

A test driver produced by an environment model is an ideal solution for real time and reactive systems. For real time systems all test cases are easily derived at generation time before the SUT begins its execution so the test driver can very efficiently provide input to the SUT at the appropriate intervals. For reactive systems, a large portion of the

[15] Perry, Dewayne and Gail Kaiser. "Adequate Testing and Object-Oriented Programming." in *Journal of Object-Oriented Programming 2(5): January/February 1990*, 14.

test cases can be generated before execution. Depending on the resources available, many of the possible branches can also be generated before execution time of the SUT.

THIS PAGE INTENTIONALLY LEFT BLANK

III. RELATED WORK

A. INTRODUCTION OF AN EVENT AND EVENT TRACE

The concept of an event and event trace is introduced by Auguston in papers on debugging automation.[16] [17] In the papers an event is defined as any action that can be detected during program execution. They also define two binary relations for events, precedence and inclusion. These relations are able to describe the execution of a program as a partially ordered set of events, or an event trace.

In a paper detailing the use of a high-level decoy specification language called CHAMELEON, Michael *et al*., detail an approach to use events and event traces to create software decoys against malicious attacks.[18] In this paper the concept of event is used to develop event patterns that are matched against program system calls to detect intrusion events. When there is an event pattern match, an action is performed. In the environment model approach an event is external to the program being tested, and the event trace is the entire set of inputs for the program.

B. USING ATTRIBUTED EVENT GRAMMAR TO MODEL AN ENVIRONMENT

Auguston et al. introduce the concept of automated testing of real-time reactive software systems based on attributed event grammar modeling of the environment in which a system will operate.[19] Here an event is defined as "any detectable action in the environment that could be relevant to the operation of the SUT." Events can have a

---

[16] Auguston, Mikhail. "A Language for Debugging Automation." in *Proceedings of Sixth International Conference on Software Engineering & Knowledge Engineering*, edited by S.K. Chang, 108-115 Skokie, Ill: Knowledge Systems Inc., June 1994.

[17] Auguston, Mikhail. "Lightweight Semantics Models for Program Testing and Debugging Automation." in *Proceedings of 7th Monterey Workshop: Modeling Software System Structures in a Fastly Moving Scenario*, 23-31. Ligure, Italy: Santa Margherita, June 2000.

[18] J. Bret Michael and others, "An Experiment in Software Decoy Design." in *Security and Privacy in the Age of Uncertainty*: *IFIP TC11 Eighteenth International Conference on Information Security*, edited by Gritzalis, D., Capitani di Vimercati, S., Samarati, P., and Katsikas, S., 253-264. (Boston, MA: Kluwer Acad. Publishers, 2003).

[19] Mikhail Auguston and others "Test Automation and Safety Assessment in Rapid Systems Prototyping." in *Proceedings of 16th IEEE International Workshop on Rapid System Prototyping* Held in Montreal, Canada, June 8-10 2005, 188-194.

precedence or inclusion relation as well no relation at all. Two events with no relation are unordered, and can even happen concurrently.

This paper by Auguston *et al*. demonstrates not only how to automatically generate and run test cases on a SUT, but additionally demonstrates how an environment model could accept and react to outputs from the SUT. This allows generated test cases to interact with the system and adjust the evolving event trace based on the results of that interaction.[20] The authors propose a method of using a large number of automatically generated tests to gain some approximation for the risk of the SUT entering into various hazardous states. By altering the probability of individual model parameters, these tests can determine the impact that parameter has on the probability of a hazardous outcome.

C. BEHAVIOR MODELS AIDING RUN-TIME VERIFICATION AND MONITORING

Behavior models based on event grammars can be designed not only for the environment, but for the SUT as well, and used for run-time verification and monitoring.[21] Auguston *et al*. state that this technique may be used to create the oracle that will allow the automation of test-result verification. So it is feasible to automate all three major phases of the software testing process: the creation of test cases, the running of test cases, and the verification of test case results.

Below are some of the advantages of using an environment model approach as taken from the two Auguston *et al*. papers:

- An environment model specified by AEG provides for automated generation of a large number of random (but satisfying the model constraints) test drivers.

---

[20] Mikhail Auguston and others "Test Automation and Safety Assessment in Rapid Systems Prototyping." in *Proceedings of 16th IEEE International Workshop on Rapid System Prototyping* Held in Montreal, Canada, June 8-10 2005, 188-194

[21] Mikhail Auguston and others, "Environment Behavior Models for Scenario Generation and Testing Automation." in *Proceedings of the First International Workshop on Advances in Model-Based Software Testing (A-MOST'05), the 27th International Conference on Software Engineering*, ACM Press (St. Louis, May 2005).

- Since the whole testing process can be automated, it becomes possible to run large numbers of test cases with which to expose errors.
- It addresses the regression testing problem: generated test drivers can be saved and reused.
- It is relatively easy to adjust the testing tool to the changing requirements by just adjusting the event grammar.
- The generated test driver contains only a sequence of calls to the SUT, external event listeners for receiving the outputs from SUT, and time delays where needed to fulfill timing constraints. Hence it is quite efficient and could be used for real-time test cases.
- Different environment models for different purposes can be designed, for example, for testing extreme scenarios by increasing probabilities of certain events. Experiments with the environment model running with the SUT provide a constructive method for quantitative and qualitative software risk assessment.
- Environment models can be designed in early stages, before the system design is complete and can be used as environment simulation tool for tuning the requirements and prototyping efforts.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DEVELOPMENT OF ENVIRONMENT MODELS

A. DIFFERENT TYPES OF ENVIRONMENT MODELS

1. Complexity of the Model

Different environment models contain varying levels of complexity both in the relationships between the events that will be modeled and how the model will interface with the SUT. Models may have very simple relationships between events. For example a simple model may have only single level of inclusions with no other dependencies between top level events. Figure 2 illustrates an example.

```
A::= B C
B::= /f1()/
C::= /f2()/
```

Figure 2. A Simple Model.

Some models may have complex relationships between various events including dependencies that require going up and down a hierarchical structure. Events may need access to other event attributes. These relations are represented by the construct of ENCLOSING. The ENCLOSING construct is discussed further in Chapter V where the specifications for the event grammar are covered.

Models may generate events that have precedence relations between them. To account for this the test generator must attach time stamps to events as they are generated, which allows the test generator to later arrange event execution by sorting events by time. Parallel events may have the same time stamp but the order of the events within the same quantum is inconsequential since by definition the events have no dependencies between them.

The relation to be established between the test driver and SUT will affect the complexity of the model as well. If the test driver is only providing inputs to the SUT, you have the simplest case of the model producing an event trace that is feed linearly as

input to the SUT. The model and test driver become more complex when the test driver must take output / feedback from the SUT and then provide the appropriate input as a reaction to the SUT output.

2. Normal and Non-Normal Behavior Represented a Model

One characteristic of a model is whether or not it is modeling a normal or non-normal environment. Normal is defined here as typical or what is expected. Models that are designed to represent a normal environment, on average, produce the proper types of inputs, in the correct sequence, at what is expected as the normal time frequency. The caveat "on average" is used because even a normal model may produce non-normal events. Due to pure randomness normal models may produce low frequency events at a higher than expected rate of occurrence, or because of interactions in the model, a normal model may produce an output that is of the incorrect type, sequence, or with the wrong timing.

Models can be designed to represent a non-normal environment. These models may produce certain events, or sequences of events at a much higher probability than what is expected in the SUT's operating environment. Also non-normal models may be designed to create timing issues that are not expected or occur at a very low frequency in the anticipated operating environment. These non-normal models are ideal tools to "stress" a SUT under varying operating conditions. A non-normal model may also be used to force certain conditions to occur at a high rate or even always occur. A fault model would provide the ideal basis from which a non-normal model could be built.

B. ENVIRONMENT MODEL CHARACTERISTICS

When determining the characteristics of an environment model one must first determine the events that occur in the environment that become an input to the SUT or must be observed by the SUT for it to carry out its functions. A calculator must take in sets of numbers and operators. A weapon selector[22] must first detect an inbound target before algorithms for weapon selection are started and a weapon selection is made.

[22] A weapon selector is a system that must decide from several locations and types of weapons, which has the best chance of intercepting an incoming missile. See Appendix A for a sample environment model.

The next consideration is what attributes the events must have in order to test the required functions of the SUT. A calculator may only be designed to take numbers of less than 6 digits. A weapon selector might need to discriminate between cruise and ballistic missiles. The last consideration when determining the characteristics of an environment model are the sequence and timing of events. Sequentially event B may always be proceeded by event A so the model must have this characteristic. Event A may occur randomly or on a set interval. Again this characteristic must be present in the model.

If it is possible that the environment may not always produce events in the correct order or even with the correct attributes the environment model must include this behavior to adequately test the SUT. An example would be the model generating event B without a preceding event A, and having event B be an integer when it should be a character.

C. CALCULATOR ENVIRONMENT MODEL CHARACTERISTICS

When developing the simple calculator environment model that would be used to test the vector calculator program, the first consideration was to determine what type of events happen in the calculator environment. It was determined that the environment consisted of a user providing an operator and then two sets of numbers to the calculator. Each of these inputs needed to be followed by an “enter” event so the inputs are discrete.

The next step was to determine the attributes of the events. For the vector calculator numbers are represented by **short** so the number acceptable from the environment is not too large, approximately 15 bits depending on the implementation. The operations acceptable are plus (+), minus (-), multiply (*) and divide (/). There is no requirement for an equal operator as the program automatically produces an output after receiving an operator and two numbers.

The sequence of the inputs from the environment was critical, as mentioned above; no output is produced unless a strict sequence of operator, number, and number is

followed. The timing of inputs is of no consequence since the calculator program has both no timing dependencies between inputs and no timing constraint in producing an output

D. WEAPON SELECTOR ENVIRONMENT MODEL CHARACTERISTICS

The theoretical weapon selector to be tested had several requirements to accomplish:

- receive an inbound missile alert
- determine the target and the time to impact of the inbound missile
- determine the priority of an inbound missile to be engaged
- decide on the best weapon to engage the inbound missile
- be able to retarget an inbound missile if a previous engagement failed to destroy it

The weapon selector would prioritize inbound missiles based on the predicted target of the enemy missile and the predicted warhead it is carrying. For example an enemy missile targeted at a large city with a nuclear warhead would receive the highest priority, while an enemy missile targeted at a military unit with a high explosive warhead would receive the lowest priority. The weapon selector would determine the warhead type by fusing intelligence information of the launch location and the flight parameters of the missile. The model environment for the weapon selector presented in Appendix A is not capable of interacting with the SUT, therefore the requirement for the weapon selector to retarget missiles is not tested.

Based on the requirements the inbound missile attributes where determined to be:

- launch location
- current location
- type (ballistic | cruise)
- payload
- target

The weapon selector environment model only needs the launch and current location attributes to generate the test cases. The other attributes are needed to allow an oracle to determine if the weapon selector met its requirements.

V. SPECIFICATION OF AN EVENT GRAMMAR

A. SOME BASIC QUESTIONS

When designing the event grammar three questions had to be answered:

- What parts of the SUT's environment needed to be represented in a model?
- How would these parts of the environment be captured by an event grammar?
- How would this model be used to automatically generate test drivers?

The first question, what parts of the SUT's environment needed to be represented in a model, is covered in the Chapter IV discussion on the development of environment models. The environment model is designed to create the proper input for the SUT. These inputs must have the appropriate attributes, be produced in a particular sequence and sent to the SUT based on certain timing constraints. Therefore the event grammar must be able to support these requirements. There is a distinction between an events timing attribute and the model's design to simulate the timing of events. For example, a model of the environment that a routing protocol must operate in could include the arrival of keep-alive messages from other routers. Let us say one of the keep-alive event's attributes is that it is sent every 30 seconds. The event should not be telling the model when to send the event. The model must determine when to send a keep-alive to the SUT, and therefore the model must be able to send or not send a keep-alive at the standard 30 second time interval.

This requirement of accurately capturing the attributes of events but ensuring the model maintains control of creation of the event trace, brings us to the second question concerning how would the essential parts of the SUT's environment be captured by an event grammar. Identifying the essential parts of the SUT's environment can be done in various ways, the most straight forward of which is, deriving the inputs required to support the SUT from the SUT's specifications and requirements. How the model maintains control over events is through a hierarchical structure. Environment models are written "top down" with the rule(s) of the model coming before and therefore controlling all event rules.

In order to automatically generate test cases using the environment model approach, the test driver generator must first be able to generate a data structure that contains the events to be modeled that occur in the SUT's environment. The model has the rules for generating these events. Some events may merely be triggers for other events to occur, while certain events will contain actions to be accomplished. The test driver generator must then traverse the data structure for events that have actions for the SUT. These events will have attributes that contain the information the SUT needs to act on in order to fulfill its requirements.

B. RELATIONSHIP SYNTAX

Figure 2 in Chapter IV illustrates the basic relationships needed between events. Events B and C are included in event A and event B precedes event C. These basic relationships are all that are need to represent event relationships in many environment models. However while modeling a range of example environments it has been determined some other relationships must be represented to accurately model some environment.

Some environments require children of rules to access attributes of the parent or other children of the same parent. The construct ENCLOSING covers this type of required link between events. See the Paderborn Shuttle Model in Appendix A for examples on the ENCLOSING construct allowing the child access to attributes of the parent event.

VI. DESIGN OF TEST DRIVER GENERATOR PROTOTYPE

A. HIGH LEVEL ARCHITECTURE

At the highest level the path to the creation of the test driver occurs after the environment model written in an AEG is translated by a compiler into the type of code needed to create a test driver for the SUT. The purpose of the compiler is to take the environment model written to test a SUT and generate a test driver. The test driver then generates the input for the SUT. The input to the SUT can be as simple as plain data presented as a flat file, as is the case of the calculator example in this thesis, or the input can be complex function calls placed within the appropriate wrappers. The test driver may also be designed to take inputs from the SUT and then provide the appropriate feedback to the SUT.

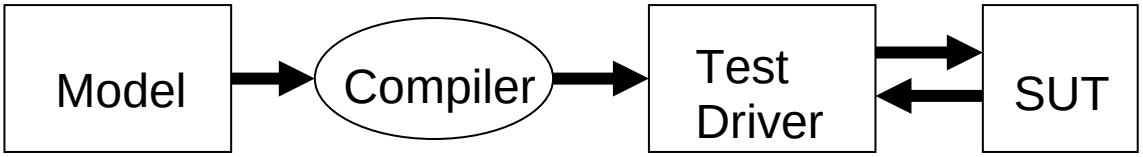

Figure 3. High Level Architecture

B. LOW LEVEL ARCHITECTURE

Refining the path that leads to the completed test driver from the environment model adds intermediate steps within the compiler and test driver. First within the complier there must be a parser that takes the environment model and parses it into an abstract syntax tree. Within the test driver several actions must be accomplished. First, from the abstract syntax tree an event trace generator must create a set of events. If required by the environment model, these events need to be sorted by timestamp to create a sorted event trace. This sorted event trace is then used by the test driver to generate the input to be used by the SUT.

It is important to note that not all events generated from the abstract syntax tree will be used by the test driver to generate code for input to the SUT. One requirement for the test generator is to traverse the tree that is produced from event trace generation and identify those events that result in actions that must be used as input to the SUT. Only these events would be used to create input for the SUT. The simplest example is an

environment event that the SUT does not observe but is in the precedence chain of an event that the SUT does observe. Take an environment model designed to test a radar system. A simple example is a missile launch event that would not be observed by a radar site on the other side of the world, but does translate to a radar target later in time. A more complex example would a model designed to produce different types of radar targets at various ranges to the radar site. Based on the size of the target and its range, there is a certain probably that the radar will detect the target, therefore the test generator must determine if the radar system will be given a radar "hit" as an input or not.

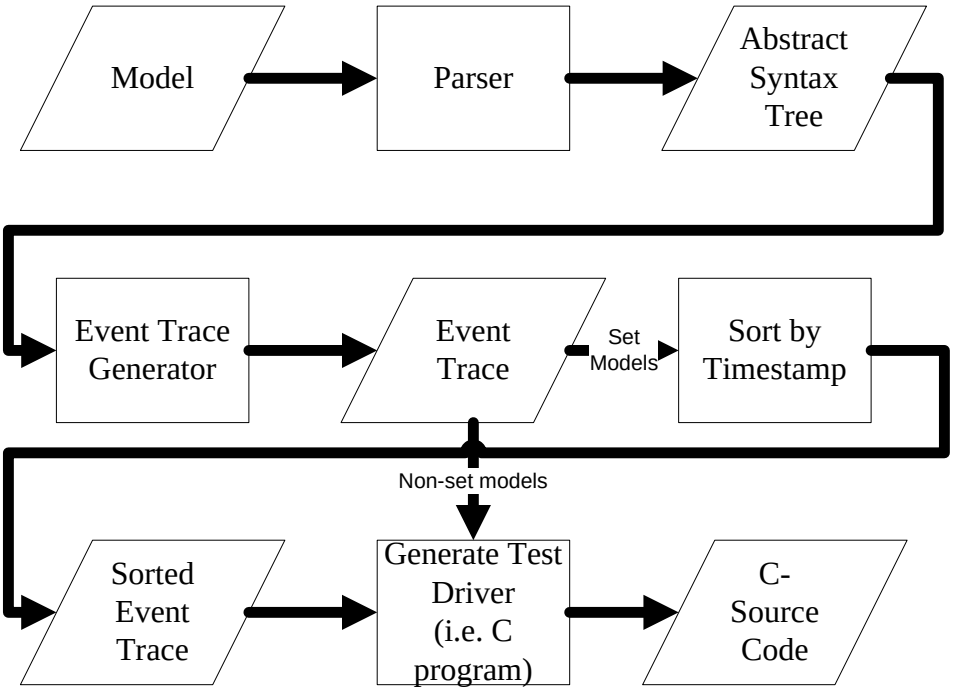

Figure 4. Low Level Architecture

C. CHOOSING A LANGUAGE TO WRITE THE PARSER AND TEST DRIVER GENERATOR IN

To write the parser and test driver generator the RIGAL programming language was chosen for its flexibility and readability. RIGAL is a programming language developed by Mikhail Auguston and Vadim Engelson as a compiler writing tool.[23] The main data structures are atoms, lists, and trees. The control structures are based on advanced pattern matching. RIGAL allows the programmer to divide a program into

[23] Mikail Auguston " Programming Language RIGAL as a Compiler Writing Tool," *ACM SIGPLAN Notices, December 1990, vol.25, #12,*.61-69.

several independent modules or rules and provides various means to tailor interaction between the modules. This gives the programmer the benefit of being able to add delete or modify rules without major modification to the whole program.

RIGAL proved to be very flexible as the parser and generator were developed and refined. As new capabilities or functions were discovered to be needed during the development phase, it was relatively easy and straight forward to modify both either the parser or generator by modifying current modules or adding new ones.

D. THE MODEL PARSER DESIGN STRUCTURE

The model parser takes a text file and uses the Rigal lexical analyzer to parse the file into individual atoms. The parser then begins normal parser tasks including putting the model into an intermediate form as a symbol rule table and reporting any syntax rule violations. Rule definitions are placed in a tree structure that preserves the hierarchical relationships between the rules. Each rule definition is itself placed in a tree structure that has the rule name as one branch and the string of atoms that makes up the rule definition on a second branch.

E. THE TEST GENERATOR DESIGN STRUCTURE

The test generator is designed to create a C program that will generate the actions to be sent to the SUT. First the intermediate form created by the parser is loaded and then the appropriate C headers are sent to the file that will be the test driver. RIGAL does not have a random number generator so a random number file created by a C program is loaded next by the generator for random number generation during execution. Now the parser output is traversed by the generator and each rule is processed to produce an event trace. Those events that produce an action are passed on as output to create the test driver.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. EXPERIMENTS

A. HYPOTHESIS

The hypothesis of the experiments was the belief that it is technically feasible to run automated tests generated from environment models on a system under test. The SUT chosen for experimentation was a vector calculator program. Tests were run with the environment model only providing normal and correct inputs and then with the environmental model allowing random or non-normal inputs.

B. TEST DESIGN

The calculator program tested was designed to ask a user for an operator (plus, minus, multiply or divide) then two numbers. Each input is received from standard in and returned to standard out. The calculator program was modified to take input from a file and direct output to a second file. The program was also modified by enclosing it inside a loop with the length of the number of test cases to be run. A copy of the modified program is in Appendix B.

To determine how the test driver needed to be structured, manual testing was done to ensure data was appropriate and correct. It was determined that a flat file with each input terminated by an end of line character would be appropriate input. Correct input depended on an operator being sent as input followed by two numbers. The test generator could have been designed to produce this flat file directly, but since more complex SUTs may require function calls with parameters as input, the generator is designed to generate a C program that when executed produces the actual test driver.

Originally the calculator environment model included up to 8 digit numbers as input. To avoid variable overflow, the model was initially modified to produce test cases with only four digit numbers since this particular calculator program uses a **short** variable to store the number input. Manual testing also allowed the identification of how would the program handle these “overloads” of the number variables. The fault model predicted the only way to generate a number that could not be represented by 15 bits from two 15 bit numbers was to use the multiplication operator. Two errors were observed

during manual testing of multiplication operations. Some "overloads" resulted in negative numbers as output. This was the result of an unrecognized integer overflow generating a now truncated binary number with the sign bit flipped to negative. The other error indication was an output that contained the first input number followed by a zero in the place for the second input number and a zero displayed as the result of the operation. This was probably the output required when an overflow condition was recognized.

C. PROCEDURES

Before each test was run, the required number of equations to be in the test case was placed in the guard of perform_calculation in the model, and in the outer loop variable of the vector calculator program. Then the parser was run on the model followed by the test driver generator. The resulting C program was run and the output file was used as the input for the calculator program. The calculator program produced a text file with the results of the equations.

D. RESULTS

The result of each test was either a text file with the number of equations run or a calculator program that had crashed.

E. ANALYSIS

The goal of the thesis was to prove that it was technically feasible to run automated tests, generated from environment models on a system under test, not to provide the method of verifying the SUTs results. However a brief ananlysis of the results did bring out some interesting points. From scanning the results the following conclusions can be made:

- the automated tests showed the result of division operation is rounded off to nearest integer, so some division results are incorrect
- the calculator program can handle division by zero without crashing
- when feed non-normal inputs the calculator program crashes

VIII. CONCLUSION

A. CONTRIBUTIONS

A “test pattern” is a tool that you can build upon and use to accomplish a specific type and scope of test. A mature test pattern has proven itself as a “best practice” way of testing for and revealing a certain kind of fault. Test patterns are used because they provide a ready-made template to examine a piece of code at a specific level and reveal the particular faults you are trying to discover. In this thesis various techniques for designing and building an environment model have been presented. The use of environment models can be seen as a use of a test pattern. Environment models can be developed and easily refined to become the mature test pattern needed for testing various systems.

The research completed has proven that it is technically feasible to run automated tests, generated from environment models on a system under test. This environment model approach to the automatic generation of software test cases is a tool that can be used in an overall testing infrastructure. The environment model approach allows testing in all types of testing from black box to white box. With a valid fault model, large number of bugs should be discovered in a short amount of time due to the automated nature of the tests.

B. FUTURE WORK

There are numerous interesting research areas that would be key extensions to the presented research on the environment model approach to automated testing. One of the extensions needed are models that have the ability to sort threads by time stamp and interact with the SUT. As mentioned in Chapter III, Auguston *et al*. have provided a solution to these problems in a recent paper so the environment model approach has this capability.

Another area for research is how the environment model approach could contribute to rapid prototyping, and risk analysis. The automated environment model approach allows easy manipulation of the parameters in the model. Manipulating one

parameter while keeping others fixed, can allow the discovery of dependencies that are not obvious. In rapid prototyping, early versions of a system can be easily tested and its "behavior" in the expected environment can be observed. This would allow early corrections to both the software code and the requisite specifications if required. In risk analysis large numbers of tests can be run and statistics built on the effects of changing individual parameters. From these statistics risk decisions could be made on where to place resources.

These and other possible capabilities when added to the environment model approach will allow this approach to tackle the complexity and scale of real world systems making it the powerful testing tool it has the potential to be.

APPENDIX A.

A. EXAMPLES OF ENVIRONMENT MODELS

1. A "Normal" Model Used to Test Vector Calculator

```
using_calculator::= ( perform_calculation )* (= 1000);

perform_calculation::=  perform_binary_calculation;

perform_binary_calculation::= press_binary_op_button enter_number  enter_number;

enter_number::= ( press_digit_button )* (<=5) press_enter;

press_digit_button::= @ genOut(RAND[0..9]) @;

press_binary_op_button::= ( p(25)  press_plus press_enter  |
                            p(25)  press_minus press_enter |
                            p(25)  press_mult press_enter  |
                            p(25)  press_div press_enter  );

press_plus::= @ genOut( "+" ) @;
press_minus::= @ genOut( "-" ) @;

press_mult::= @ genOut( "*" ) @;

press_div::= @ genOut( "/" ) @;

press_enter::= @ press_enter( "" ) @;
```

2. A “Non-Normal” Model for a Calculator

```
using_calculator::= ( perform_calculation )* (= 5);

perform_calculation::=  perform_binary_calculation;

perform_binary_calculation::= ( p(50) press_binary_op_button |
                                  p(50) enter_number )
                                  enter_number
                                ( p(50) press_binary_op_button |
                                  p(50) enter_number );

enter_number::= ( press_digit_button )* (<=5) press_enter;

press_digit_button::= @ genOut(RAND[0..9]) @;

press_binary_op_button::= ( p(25)  press_plus press_enter  |
                                p(25)  press_minus press_enter  |
                                p(25)  press_mult press_enter  |
                                p(25)  press_div press_enter  );

press_plus::= @ genOut( "+" ) @;

press_minus::= @ genOut( "-" ) @;

press_mult::= @ genOut( "*" ) @;

press_div::= @ genOut( "/" ) @;

press_enter::= @ press_enter( "" ) @;
```

3. A Complex Model for a Calculator

```
using_calculator::= (perform_calculation)* (<= 100);

perform_calculation::= ( p(20) perform_unary_calculation |
                         p(80) perform_binary_calculation );

perform_unary_calculation::= enter_number press_unary_op_button;

perform_binary_calculation::=
   ( p(80) ( enter_number press_binary_op_button enter_number press_equal ) |
     p(20) ( press_left_par perform_calculation*     press_right_par));

enter_number::= ( press_digit_button )* (<=8) press_enter;

press_digit_button::= @ genOut ( RAND[0..9] ) @;

press_unary_op_button::= ( p(60) @ press_negation() @ | p(20) @ press_sin() @ |
                           p(20) @ press_cos() @ );

press_binary_op_button::= ( p(25) @ press_plus() @ | p(25) @ press_minus() @ |
                            p(25) @     press_multiply() @ | p(25) @ press_divide() @ );
```

4. A Model for a Weapon Selector

In this model a weapon selector is a system that must decide from several locations and types of weapons, which has the best chance of intercepting an incoming missile.

launch::= (boost_phase ascent_phase terminal_phase)* (<=100)

boost_phase::= boost_1 boost_2 boost_3

boost_1::= / send_flash() /

boost_2::= (/ calculate_coordinates() / [p(0.7) /send_radar_hit()/])* (<=10)

boost3::= (/ calculate_coordinates() / [p(0.9) /send_radar_hit()/])* (<=10)

ascent_phase::= (/ calculate_coordinates() / [p(0.9) /send_radar_hit()/])* (<=10)

terminal_phase ::= (/calculate_coordinates()/ [p(0.9) /send_radar_hit()/]
)* (<=10)
if coordinates == target_location /send boom_signal()/

5. Model for the Paderborn Shuttle System

The following model is found in Auguston *et al*. paper on "Environment Behavior Models for Scenario Generation and Testing Automation". It is presented here as an example to illustrate the ENCLOSING construct, which provides access to the attributes of parent events.

```
Shuttle_system:     {* Shuttle *} (1..ShuttleNum);

Shuttle:            /Shuttle.id = Unique_num;
                     Shuttle.at_station = Rand(1..StationNum);
                     Shuttle.account = MaxAccount;
                     Shuttle.limit = 0;
                    Shuttle.retired = false;/
                    (*      WAIT order(Shuttle.start, Shuttle.destination)
                    WHEN (Shuttle.start == Shuttle.at_station)
                            ( /send_offer(Shuttle.id,
                                    calculate(Shuttle.start, Shuttle.destination);/
                            WAIT confirmation(Shuttle.accepted)
                            WHEN (Shuttle.accepted) Move
                            )
                    *);

Move:
        WHEN (ENCLOSING Shuttle.limit > MaxLimit)
                Maintenance
        / ENCLOSING Shuttle.at_station = next_station(ENCLOSING Shuttle.at_station,
                                                      ENCLOSING Shuttle.destination);
        ENCLOSING Shuttle. account -= TransitFee;
        ENCLOSING Shuttle.limit += Wear;
        send_notification(ENCLOSING Shuttle.id, ENCLOSING Shuttle.at_station );/
        CheckAccount
        WHEN (ENCLOSING Shuttle. at_station == ENCLOSING Shuttle. Destination)
        /ENCLOSING Shuttle. account += Payment;/
        ELSE Move;

Maintenance:
        / ENCLOSING Shuttle.account -= MaintenanceFee;
          ENCLOSING Shuttle.limit = 0;/
          CheckAccount;

CheckAccount:
        WHEN (ENCLOSING Shuttle. Account <= 0)
                (/ENCLOSING Shuttle.retired = true;/
                  BREAK);
```

B. EXAMPLES OUTPUTS

1. Example Parser Output

```
<<<=EXITS FROM RULE #program: SUCCESS
RESULT=

  <.base:'using_calculator',
    rule_table:
    <.using_calculator:
      (.
        <.type:'iteration',
          pattern_list:(.<.type:'rulename',name:'perform_calculation'.> .),
          guard:<.op:'=',numb:1000.>
        .>
      .)
      ,
      perform_calculation:(.<.type:'rulename',name:'perform_binary_calculation'.> .),
      perform_binary_calculation:
      (.<.type:'rulename',name:'press_binary_op_button'.>
<.type:'rulename',name:'enter_number'.> <.type:'rulename',name:'enter_number'.>
      .)
      ,
      enter_number:
      (.
        <.type:'iteration',
          pattern_list:(.<.type:'rulename',name:'press_digit_button'.> .),
          guard:<.op:'<=',numb:5.>
        .> <.type:'rulename',name:'press_enter'.>
      .)
      ,
      press_digit_button:
      (.
        <.type:'action',
          funtion_name:'genOut',
          arg_list:(.<.type:'random_num',limit1:0,limit2:9.> .)
        .>
      .)
      ,
      press_binary_op_button:
      (.
        <.type:'alternative',
          alt_list:
          (.
            <.probab:25,
              pattern_list:
              (.<.type:'rulename',name:'press_plus'.> <.type:'rulename',name:'press_enter'.>
              .)
```

```
        .>
        <.probab:25,
          pattern_list:
          (.<.type:'rulename',name:'press_minus'.> <.type:'rulename',name:'press_enter'.>
          .)

        .>
        <.probab:25,
          pattern_list:
          (.<.type:'rulename',name:'press_mult'.> <.type:'rulename',name:'press_enter'.>
          .)

        .>
        <.probab:25,
          pattern_list:
          (.<.type:'rulename',name:'press_div'.> <.type:'rulename',name:'press_enter'.>
          .)

        .>
      .)

    .>
  .)
  ,
  press_plus:(.<.type:'action',funtion_name:'genOut',arg_list:('+' .).> .),
  press_minus:(.<.type:'action',funtion_name:'genOut',arg_list:('-' .).> .),
  press_mult:(.<.type:'action',funtion_name:'genOut',arg_list:('*' .).> .),
  press_div:(.<.type:'action',funtion_name:'genOut',arg_list:('/' .).> .),
  press_enter:(.<.type:'action',funtion_name:'press_enter',arg_list:(" .).> .)
 .>
.>
```

2. Example “Normal” Test Driver Generation Program

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
// ofstream constructor opens the file
ofstream genOut( "driver_output.txt", ios::out );
genOut  <<  "-" ;
genOut << endl ;
genOut  <<  "7";
genOut  <<  "1";
genOut << endl ;
genOut  <<  "6";
genOut  <<  "6";
genOut  <<  "4";
genOut  <<  "8";
genOut << endl ;
genOut  <<  "*" ;
genOut << endl ;
genOut  <<  "0";
genOut << endl ;
genOut  <<  "0";
genOut << endl ;
genOut  <<  "*" ;
genOut << endl ;
genOut  <<  "5";
genOut << endl ;
genOut  <<  "0";
genOut  <<  "0";
genOut << endl ;
genOut  <<  "/" ;
genOut << endl ;
genOut  <<  "2";
genOut  <<  "8";
genOut  <<  "2";
genOut  <<  "1";
genOut << endl ;
genOut  <<  "7";
genOut << endl ;
genOut  <<  "+" ;
genOut << endl ;
genOut  <<  "2";
genOut << endl ;
genOut  <<  "8";
genOut << endl ;
 return 0; }
```

3. Example “Normal” Test Driver

```
-
71
6648
*
0
0
*
5
00
/
2821
7
+
2
8
```

4. Example “Non-Normal” Test Driver Program

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
// ofstream constructor opens the file
ofstream genOut( "driver_output.txt", ios::out );
genOut  <<  "*" ;
genOut << endl ;
genOut  <<  "1";
genOut  <<  "6";
genOut  <<  "6";
genOut  <<  "6";
genOut << endl ;
genOut  <<  "1";
genOut  <<  "5";
genOut << endl ;
genOut  <<  "/" ;
genOut << endl ;
genOut  <<  "7";
genOut  <<  "1";
genOut << endl ;
genOut  <<  "0";
genOut  <<  "0";
genOut << endl ;
genOut  <<  "2";
genOut  <<  "8";
genOut  <<  "2";
genOut  <<  "1";
genOut << endl ;
genOut  <<  "7";
genOut << endl ;
genOut  <<  "/" ;
genOut << endl ;
genOut  <<  "-" ;
genOut << endl ;
genOut  <<  "2";
genOut << endl ;
genOut  <<  "4";
genOut  <<  "0";
genOut << endl ;
genOut  <<  "/" ;
genOut << endl ;
genOut  <<  "2";
genOut  <<  "4";
```

```
genOut  <<  "6";
genOut << endl ;
genOut  <<  "5";
genOut  <<  "2";
genOut  <<  "8";
genOut  <<  "5";
genOut << endl ;
 return 0; }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B.

A. SOURCE CODE

1. Environment Parser

```
-- MODEL PARSER version 6
-- James Imanian
-- last modified 23MAY05
--
-- takes attributed event grammar model and generates the
-- intermediate form needed by test generator


-----------------------------------------------------------------
#main
   -- take environment model file named CalcEv3.txt and parse
   $lexems:= #CALL_PAS(35 'CalcEv3.txt' 'L+A-U-P-C+p-m+'); --C lexar

   OPEN MSG ' ';
   $var := #program($lexems);

   -- intermediate form for parser input will be "calctree"
   SAVE $var 'calctree';

##


#program
   (. (* $l !.:= #ruledef  ';' *)  .)
   /FORALL $e IN $l DO
      $rule_table++:= <. $e.rulename: $e.pattern_list.>
      OD;

   RETURN
      <. base:       $l[1].rulename,
        rule_table: $rule_table
      .>
   /

##


#ruledef
   -- look for signature of a rule definition
   $Id  ':' ':' '=' (* $plist !.:= #pattern *)
```

```
   /RETURN <. rulename:    $Id,
             pattern_list: $plist .> / ;;

   -- if rule does not properly terminate with a semicolon
   (* $l!.:= S' ($$<>';') *)
   / MSG << syntax error in rule $l /

##

-- go through $plist and determine what type of rule it is
#pattern
   $p:=(   #action
         ! #alternative
         ! #rulename
         ! #iteration )
       / RETURN $p /

##


#rulename
   $Id
   / RETURN <. type: rulename,
             name: $Id .> /
##


#iteration
   '(' (* $p!.:= #pattern *) ')' '*' [ $guard:= #guard ]
   / RETURN <. type:        iteration,
             pattern_list: $p,
             guard:        $guard .> /


##


#guard
   '(' ( $op:= '=' ! ( '<' '='/ $op:= '<=' / ) )
   ( '0' / $num:=0 / ! $num:= ( #NUMBER ! #random ) ) ')'
   / RETURN <. op:   $op,
             numb: $num .> /

##
```

```
#action -- always a function call
    '@' $Id '(' (* $arg_list!.:= ( #NUMBER ! #random ! $ATOM ) * ',' )
        ')' '@'
    / RETURN <. type:        action,
               funtion_name: $Id,
               arg_list: $arg_list .> /

##


#alternative
       '(' (* 'p' '(' $num:= #NUMBER ')' (* $p!.:= #pattern *)
          / $alt_list!.:= <. probab: $num,
                            pattern_list: $p .>;
            $sum+ := $num;
            $p:= NULL;
          /
          * '|' ) ')'


       / IF $sum <> 100 -> MSG << Sum of prob not 100 and is $sum FI;
         RETURN <. type:     alternative,
                  alt_list: $alt_list .> /


##


-- rule that will pass to generator a request (in a tree form)
-- for a random number within a range
#random
   -- RIGAL lexical analyzer does not recognize zero as a number
   -- so zero character must be assigned to number zero
   'RAND' '[' ('0' / $num1:=0 / ! $num1:= #NUMBER) '.' '.'
           ('0' / $num2:=0 / ! $num2:= #NUMBER) ']'
   / RETURN <. type:   random_num,
              limit1:  $num1,
              limit2:  $num2  .> /

##
```

2. Test Case Generator

```
-- GENV8.RIG Generator version 8
-- James Imanian
-- Last modified 30MAY05
--
-- takes intermediate form produced by model parser
-- and produces an event trace
--
-- loads random number file "randNums.txt" for use in
-- random number generation

-------------------------------------------------------

-- The main rule will generate the skeleton of the test driver
#main
   LOAD $tree 'calctree';
   PRINT $tree;

   --globals
   $rule_table:= $tree.rule_table;

-------------------------------------------------------

   -- create a file that will be the test driver
   OPEN gen 'result.c';

   -- give the file the needed headers
   gen << '#include <iostream> ';
   gen << '#include <fstream> ';
   gen << 'using namespace std; ';


   gen << 'int main() { ';

   -- send outputs to a file "driver_output.txt"
   gen << '// ofstream constructor opens the file ';
   gen << 'ofstream genOut( "driver_output.txt", ios::out ); ';

   -- load random number table
   $random_table:= #CALL_PAS( 35 'randNums.txt' 'L+A-U-P-C+p-m+');

   -- $current_random will be used as a pointer allowing the
   -- the progam to cycle through randNums.txt multiple times
   -- if needed
   $current_random:= 0;
```

```
   -- trace rule will traverse tree generated by the parser
   -- and produce the events needed for test generator
   $trace:= #generate_from_rule($tree.base);

   -- properly end the test generator program
   gen<< ' return 0; }'

##


#generate_from_rule
   -- initialize a variable
   $rule_name

   -- use of LAST allows a rule to look to parent rule(S)
   -- and use a variable there
   / $plist:=LAST #main $rule_table.$rule_name;
     FORALL $p IN $plist DO
        #generate_from_pattern($p)
     OD
   /

##

#generate_from_pattern
   -- iteration
   <. type: iteration,
      pattern_list: $plist,
      [guard: <. op: $op,
              numb: $limit
           .>]
   .>

   -- determine what type of iteration is to be done
   -- iterations are either a set number or less than or equal
   -- to a set or random number
   / IF $op = '=' ->
        -- Ability to detect guard, and ensure random number of
        -- reps is withn range via a modulo sceme
        IF #NUMBER( $limit ) ->
           $numb_of_repetitions:= $limit
        ELSIF T -> $numb_of_repetitions:= #get_rand( $limit)
        FI;

     ELSIF    T  ->
         IF #NUMBER ($limit) ->
```

```
            $numb_of_repetitions:= #random() MOD ( $limit + 1 )
         ELSIF T ->
            $x:= #get_rand( $limit );
            $numb_of_repetitions:= #random() MOD ( $x + 1 )
         FI;
      FI;

      -- If MOD function produces numb of reps = 0
      IF $numb_of_repetitions = 0 -> $numb_of_repetitions:=1;
      FI;

   -- need to determine if more iterations need to be completed

   $count:= 0;

   LOOP
      IF $count >= $numb_of_repetitions -> RETURN NULL;
      FI;
      FORALL $p IN $plist DO
         #generate_from_pattern($p);
      OD;
      $count +:=1;

   END -- loop

   /;;




-- alternative
   <. type: alternative,
     alt_list: $L
   .>

   / -- make the decision which alterntive to use

   -- first try alternatives one by one.  if probabilty of alternative
   -- plus previous alternatives equals the random number then execute
   -- that alternative

   $determined_prob:= #random();

   FORALL $a IN $L DO
      $sum +:= $a.probab;
      IF $determined_prob <= $sum ->
```

```
        FORALL $p IN $a.pattern_list DO
           #generate_from_pattern($p)
        OD;
        RETURN NULL;
      FI
   OD;

   -- enforce the last alternative

   $a:=$L[-1];
   FORALL $p IN $a.pattern_list DO
    #generate_from_pattern($p)
   OD;

   /;;

-- action
   <. type:         action,
     funtion_name:  $Id,
     arg_list:      (. (* $arg_l!.:= ( #NUMBER ! #get_rand ! $a) *) .)
   .>

   /
   IF $Id = 'press_enter'->
          gen<< 'genOut << endl'

   ELSIF T -> gen<< $Id ' << ';
      FORALL $e IN $arg_l DO
        IF #NUMBER( $e )->
          gen<] @ "" #IMPLODE($e) ""
        ELSIF ( $Id <> 'press_enter') ->
                  gen<] @ "" #IMPLODE($e) "' ' ;
        FI;
      OD;
   FI;

   -- action will result in line of code so it must be terminated with ;
   gen<] ';' ;

   /;;


-- rule name
   <. type: rulename,
     name: $Id .>
```

```
   / #generate_from_rule($Id) /

##


#random
   -- enter random number file, return the next random number
   -- and increment pointer
   /LAST #main $current_random:= LAST #main $current_random MOD
                              #LEN( LAST #main $random_table) +1;

    RETURN LAST #main $random_table [ LAST #main $current_random ] /

##


#get_rand
   <. type:  random_num,
            limit1:  $num1,
            limit2:  $num2  .>

   / $x:= $num2 - $num1;
     -- ensure the result is returned as a positive number
     IF $x < 0 ->
         $x:= -$x
     FI;

     RETURN $num1 + #random() MOD $x /

##
```

3. Modified Vector Calculator Program

```
//*************************************
//
// Name: Simple Basic Calculator
// Description:Here is a simple calculator I built with the use of vectors.
// Please vote.
// By: GamaNetwork
//
// Assumes:Calculator that uses simple vectors
//
// This code is copyrighted and has// limited warranties.  Please see http://
//www.PlanetSourceCode.com/vb/scripts/ShowCode.asp?txtCodeId=9255&lngWId=3//
// for details.
 //
//
// Code has been modified for testing.  It now takes inputs from a file “driver_output.txt”
// and outputs to file “calc_output.txt”
//
// Comment lines beginning with JAI are comments of the tester James Imanian
//

#include <iostream>
#include <vector>
#include <string>
#include <stdlib.h>
#include <fstream>
using namespace std;
using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::ifstream;
int main()

{
   // JAI Modification to open input file

   // ifstream constructor opens the file
   ifstream inCalcFile( "driver_output.txt", ios::in );

   // exit program if ifstream could not open file
   if ( !inCalcFile ) {
   cerr << "File could not be opened" << endl;
   exit( 1 );
   } // end if
   // JAI end of testing modification
```

```
vector<short> alpha(2);
short i;
short r1;
string typ;
char c1="";
char c2='=';

// JAI input not needed from user
//cout<<"|-------------------------|"<<endl;
//cout<<"| - Gamanets Simple Calc - |"<<endl;
//cout<<"| lx_8000@hotmail.com |"<<endl;
//cout<<"|-------------------------|"<<endl;
//cout<<"Enter Basic Math Type:"<<endl;
//cin>>typ;

// JAI for loop for multiple test cases added for testing
for(int j=0; j<1000; j++){


   // JAI modification to read from input file
   inCalcFile>>typ;



   for(i=0;i<2;i++)
   {
      // JAI testing modification
         //cout<<"\nValue Numbers: "<<i+1<<":";
      //cin>>alpha[i];
         inCalcFile>>alpha[i];
   }

   if(typ=="+")
   {
      r1=alpha[0] + alpha[1];
   }
   if(typ=="-")
   {
      r1=alpha[0] - alpha[1];
                }
      if(typ=="*")
   {
      r1=alpha[0] * alpha[1];
```

```
        }

        if(typ=="/")
        {
           r1=alpha[0] / alpha[1];
        }

        // JAI output to be directed to a file


//cout<<""<<alpha[0]<<""<<typ<<""<<alpha[1]<<""<<c2<<""<<c1
        //<<""<<r1<<""<<c1<<""<<endl;

        // JAI Modification to open output file

        // ofstream constructor opens the file

        // NOTE program APPENDS the file.
        // ERASE file if you do not want old results.

        ofstream outCalcFile( "calc_output.txt", ios::app );

           // exit program if ofstream could not open file
           if ( !outCalcFile ) {
              cerr << "File could not be opened" << endl;
              exit( 1 );

           } // end if

           // JAI end modification


outCalcFile<<""<<alpha[0]<<""<<typ<<""<<alpha[1]<<""<<c2<<""<<c1
                <<""<<r1<<""<<c1<<""<<endl;

    } // JAI end test for

    // JAI pause not needed
    //system("Pause");
    return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Auguston, Mikhail. "A Language for Debugging Automation." In *Proceedings of Sixth International Conference on Software Engineering & Knowledge Engineering*, edited by S.K. Chang, 108-115 Skokie, Ill: Knowledge Systems Inc., June 1994.

Auguston, Mikhail. "Lightweight Semantics Models for Program Testing and Debugging Automation." In *Proceedings of 7th Monterey Workshop: Modeling Software System Structures in a Fastly Moving Scenario*, 23-31. Ligure, Italy: Santa Margherita, June 2000.

Auguston, Mikhail, James Bret Michael, Man-Tak Shing. "Environment Behavior Models for Scenario Generation and Testing Automation." In *Proceedings of the First International Workshop on Advances in Model-Based Software Testing (A-MOST'05), the 27th International Conference on Software Engineering*, ACM Press St. Louis, May 2005.

Auguston, Mikhail, James Bret Michael, and Man-Tak Shing. "Test Automation and Safety Assessment in Rapid Systems Prototyping." In *Proceedings of 16th IEEE International Workshop on Rapid System Prototyping*, 188-194. Montreal, Canada, June 8-10 2005.

Bennett, Simon, John Skelton, and Ken Lunn. *UML*. New York: McGraw-Hill, 2001.

Binder, Robert. *Testing Object-Oriented Systems.* Boston, MA: Addison-Wesley, 2000.

Boehm, Barry., "Verifying and Validating Software Requirements and Design Specifications," In *IEEE Software 1 January 1984*, 75-88.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The UML Users Guide*. Reading, MA: Addison-Wesley, 1999.

Boyapati, Chandrasekhar, Sarfraz Khurshid and Darko Marinov. "Korat: Automated Testing Based on Java Predicates," In *Proceedings of ACM International Symposium on Software Testing and Analysis, July 2002*, 123-133.

Kansomkeat, Supaporn, and Wanchai Rivepibon, "Automated-Generating Test Case Using UML Statechart Diagrams," In *Proceedings of SAICSIT 2003*, 296-300.

Michael, J. Bret, Georgios Fragkos, and Mikhail Auguston. "An Experiment in Software Decoy Design." In *Security and Privacy in the Age of Uncertainty*: *IFIP TC11 Eighteenth International Conference on Information Security*, edited by Gritzalis, D., Capitani di Vimercati, S., Samarati, P., and Katsikas, S., 253-264. Boston: Kluwer Acad. Publishers, 2003.

Mosley, Daniel, and Bruce Posey. *Just Enough Software Test Automation*. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

Perry, Dewayne and Gail Kaiser. “Adequate Testing and Object-Oriented Programming.” In *Journal of Object-Oriented Programming 2(5)* 13-19, January/February 1 990.

Schach, Stephen. *Object-Oriented and Classical Software Engineering*. Boston, MA: McGraw Hill, 2002.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Mikail Auguston
   Naval Postgraduate School
   Monterey, California

4. Professor Bret Michael
   Naval Postgraduate School
   Monterey, California