Formal Approaches to Software Testing

P. Dasiewicz
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario Canada N2L 3G1
email: dasiewic@swen.uwaterloo.ca

Abstract

The process of testing software is an important technique for checking and validating the correctness of software. Unfortunately, it is usually difficult, expensive, time consuming and often error prone to achieve both an effective and efficient testing process. Formal methods are a method of specifying and verifying software systems using mathematical and logic approaches. This allows the analysis and reasoning of software systems with precision and rigor. Formal methods target the verification and the proving of correctness, while testing can only show the presence of errors. The use of formal methods can also automate the generation of test cases from formal specifications which can lead to less expensive and less error prone testing process.

Keywords: *software testing, model checking, UML statecharts, software component testing.*

1. INTRODUCTION

It is generally accepted that verification techniques such as model checking are best applied in the early stages of system design when the costs are relatively low and the potential benefits are high. Formal methods applied to models of component interactions can simplify testing of component interactions by formally defining testing requirements for interactions, and automatically generating required test cases using model-checking technology. This approach allows more rigorous testing, since formal models and test selection criteria do not require individual interpretation by testers.

Model checking has been successfully applied to the verification of hardware systems. For a given transition system and property to be checked, model checking technology will exhaustively search the state space of the input transition system to determine whether the given property holds. The success in hardware verification has been partially due to many innovative data structures, especially Binary Decision Diagrams (BDDs) which are capable of encoding Boolean functions in a highly compact form. However, efficient encoding for Boolean domains is usually not as useful for software systems as it is for hardware systems.

An alternative approach is to use state-space exploration for software systems written with general purpose programming languages such as C, C++, or Java without manually constructing models. In these approaches, the source code (usually a subset) is translated into the modeling language of the model checker. Using automatic translation makes it very difficult to yield tractable models when all language features and standard libraries are included. As an alternative to state recording model checking, VeriSoft[21] performs stateless model checking of C++ programs but is limited to checking safety properties. Similar approaches used for Java[22] are also limited to checking safety properties. Other approaches employ runtime analysis which attempts to conclude properties of a program from a single observed execution trace of the program[20]. For proprietary software components, the source code may not be available limiting the effectiveness of these approaches. These approaches appear to be more suitable for general determination of safety properties rather than being test requirement driven.

Proceedings of the 2002 IEEE Canadian Conference
on Electrical & Computer Engineering
0-7803-7514-9/02/$17.00 © 2002 IEEE

Current methods are not feasible for testing complex component interactions. They either require too much manual effort for modeling and testing, or create models that are much too complex for analysis. To be feasible, two major obstacles must be overcome: the state explosion problem and the cost of creating formal models of software components suitable for testing.

There are many advantages and proposals to apply finite-state verification technologies (e.g. model checking) at the design level. This is attractive since formal methods are more tractable at this level due to the reduced complexities and the potential to reduce maintenance costs by finding any errors early in the development cycle. A major deficiency of verification at the design level is that many errors occur at the source level despite verified designs. These errors are introduced at levels of detail that designs do not deal with (assuming designs exist in the first place).

The Unified Modeling Language (UML) is a simple, yet powerful, general-purpose visual modeling language that is used to specify, visualize, construct and document the artifacts of a software system. The UML is a semi-formal language, since its syntax and static semantics are defined precisely, but its dynamic semantics are not specified formally. Component interfaces and protocol specifications can be modeled in different ways within UML. UML *sequence diagrams* can be useful in describing a specific interaction scenario but may require a large number (perhaps infinite) of these diagrams to completely specify the interaction of a complex component with its clients. A more concise, and compact, representation of these scenarios is to use a UML *statechart diagram* as the basis for a dynamic, behavioral description. The UML statechart diagram is the focal point of our proposed research for interaction test case generation. Various approaches to date address either unit level or integration level test case generation without targeting the more difficult component interaction testing problem.

The UML has become accepted as the standard notation for the design of object-oriented software systems. We overview the application of model checking technology (i.e. using SPIN, SMV, SMC) to the validation and test case generation of software systems modeled with UML statecharts and interaction diagrams. Applying model checking technology to software systems at the code level also posses many unique problems and potential limitations. In particular, we examine the applicability of formal methods (and specifically model checking) in the area of component integration and interaction testing based on UML behavioral models and our own ObjectState[24] based component interaction testing methodology.

2. Integration/Component Level Testing

The recent research by Offutt[8] is of particular interest. Offutt generates software system-level test cases for components specified with UML statechart diagrams. The research is restricted to transitions in the component state caused by *change events*. These event types model an event that occurs when a specified Boolean expression becomes true due to the change in value of one or more attributes or associations. These are used for generating test cases since change events can be expressed as predicates. Change event enabled transitions are used to define various levels of testing coverage such as transition coverage, full predicate coverage and transition-pair coverage.

A "proof-of-concept" tool, UMLTEST, was developed and integrated with the Rational Rose case tool. The major assumptions of UMLTEST are that all transitions are triggered by change events, all events and conditions can be expressed via Boolean type class attributes, and that all transitions are deterministic. Empirical results from using the tool on several small software components (order of 400 lines of C) indicate that highly effective tests can be automatically generated for system-level specification based testing.

Offutt also proposes a solution for a most difficult part of test case generation: to determine the test prefix, which includes all inputs necessary to ensure that the system is in some particular pre-state before the test case is run. The work has some limitations since test cases were only generated from statecharts using enabled transitions. Since other types of transitions were not used, certain states may never be entered. The model used only Boolean variables but could possibly be extended to use enumerated integer data types as well. While Offutt did not address the interaction testing problem, the test prefix generation is of particular interest and may be extended for other applications. Offut has recently extended his work by incorporating UML collaboration diagrams for static checking and test generation[9].

The research by Yoon[10] on UML-based test case generation for component integration is also of interest. Their approach differs since they base their test

generation solely on UML sequence and collaboration diagrams rather than statecharts.

The UML test model consists of the node which represents both the integration target and the message flow representing the interaction between the nodes. The node represents the state of one Atomic System Function[11] (ASF) while the message flow expresses the message that produces ASFs.

The testing technique proceeds to first extract the sequence diagram that shows the normal and abnormal sequences for each flow of event. In the case of concurrency, the collaboration diagram for each flow of event is also extracted. At that point, the sequence and collaboration diagrams are divided into ASF units determined by the message transitions that occur between the components. Next, the node and message flows are extracted from the ASFs. Finally, the test cases are obtained by applying the test case criteria (usually all-edge selection to cover all edges in the graph).

The sequence diagrams (and possibly the collaboration diagrams) are the sole input for their UML test model. After the test model is converted into a graph, all-edge test case selection is applied to obtain the test cases. This approach has some limitations. Sequence diagrams may be sufficient in describing a specific interaction scenario, but it may often require a large number of sequence diagrams to specify the interactions of a complex component. Test cases are only produced for the specified normal (or abnormal) sequence diagrams.

Kim[12] reports an alternative approach for unit (component) level test case generation. UML statecharts are converted into extended finite state machines (EFSMs) to flatten the hierarchical and concurrent structure of states and to eliminate broadcast communications. Control flows in UML state diagrams are identified in terms of paths in the EFSMs after a transformation into flow graphs. They propose a set of coverage criteria: path coverage, state coverage and transition coverage for test case generation. Test cases for these criteria can be constructed by simple breadth or depth first searches over the EFSMs.

Test cases, based on data flow from UML statecharts, can also be generated by applying conventional data flow analysis techniques such as *all-definition*, *all-use* and *all def-use paths*. By applying the data-flow techniques and coverage criteria, test cases are generated as a set of paths that cover the associations between the definitions and uses of each variable and state within the UML statechart diagrams. The test cases determine whether class implementations provide the correct control and data flow as specified in the specification. The work is based on unit testing of classes and does not consider the interrelationships (interaction) between classes.

3. System/Integration Testing

Hartmann[7] proposes a UML-based integration testing approach when testing the interfaces of several components. UML statecharts are considered as Mealy finite state machines with a restricted point-to-point synchronous communication model. A simple composition of state machines can easily cause an exponential increase in the states of the resultant machine with many unreachable states that must be removed later. Hartmann describes an incremental composition and reduction algorithm using reachability computations. Their test generation relies on the conventional category-partition method where states are the equivalence classes and are represented by partitions. For integration test cases, only those transitions that involve component interactions are chosen. Some limitations include: component interactions are modeled as synchronous communications; event exchanges contain no parameters or values; no support for nested machines with internal data conditions affecting the global state; no timing constraint support. The method generates test cases from an integration point of view where the basic test requirement is to exercise all interfaces rather than to perform component interaction testing with specific test requirements to expose explicitly chosen behavioral characteristics.

4. Source Code Based Testing

A major disadvantage with applying standard model checking technology at the code level is the state space explosion problem where the state space grows exponentially as the number of system components increases[23]. This is a lesser problem at the design level since designs usually contain much less detail. As such, basic model checking approaches do not scale well at the source code level and typically are limited to small programs in the range of 10 – 20K lines of code. Automatically checking the resulting transition systems for correctness properties is quite expensive and hence the necessity to reduce the state space as much as possible. For a given correctness

property to be verified, much of the original source code is not required. Eliminating this irrelevant source code can reduce the size of the resultant transition system.

One such approach proposed by Hatcliff [2] applies program slicing methods to reduce the resulting transition system. This slicing technique plays a major role in Bandera [3] which extracts finite-state machine models from Java source code. Essentially, Bandera takes Java source code and generates a program model in the input language of standard verification tools, e.g. SPIN[4], SMV[5]. Bandera's front-end translates the Java source into Jimple, a high level intermediate language. Jimple was introduced in Soot[6] as the target language of a Java decompiler. Jimple is a language of control-flow graphs where statements appear in three-address-code form. The slicer uses the user provided correctness property (specified in temporal logic form) to slice away irrelevant Jimple program components. Bandera's back-end uses this reduced system to generate a finite-state model in the selected verification tool's input language. Bandera interprets the verifier output, and when the specified property fails to hold, the verifier specific counter example is mapped back to the original Java source code.

The Java Path Finder (JPF), a verification and testing environment for Java, has been developed by Visser[1]. Their approach integrates the areas of model checking, program analysis and testing into one tool. The goal was to apply formal methods (i.e. model checking) to programs at the source level. This required the construction of a new Java Virtual Machine to interpret the bytecode. The JPF integrates state compression for handing of large state spaces, and applies partial order reduction, slicing, predicate abstraction[13], and runtime analysis to reduce the state space. A custom model checker was developed that could execute all the bytecode instructions. The model checker was an explicit state model checker similar to Spin, rather than a symbolic one based on Binary Decision Diagrams (BDDs) such as SMV[14]. Initially, the model checker was limited to checking for safety properties such as deadlocks, invariants and user defined assertions in the code. Their goal was to add temporal logic model checking in the near future. The overall approach does not appear to scale well and may be initially limited to critical sections of program source code.

JPF's runtime analysis was based on the concept of executing the program once and observing the execution trace. This information is then used to predict whether other execution traces *might* violate some properties of interest. The observed trace need not already violate any properties of interest. The algorithms do not guarantee that errors will be found and may yield false positives but their attraction lies in their scalability for large systems. One such algorithm is a data race detection algorithm, Eraser[15]. A data race can occur when a shared variable is accessed by two concurrent threads, at least one access is a write, and the treads use no explicit mechanism to prevent simultaneous access. JPF also used a custom locking order analysis algorithm called *LockTree*. The LockTree algorithm determines potential deadlocks by detecting differences in the order in which threads access locks. An ordering example would be if one thread accesses two locks L1 and then L2 (in that order) while another thread accesses the locks in reverse order. The deadlock condition might arise if the first thread accesses L1 first and the second thread takes L2. If both locks are held until each thread attempts to take its second lock, then deadlock will exist. Given this simple definition, then a program is deadlock free if all locks are accessed in the same order. The LockTree algorithm searches for the violation of this ordering between locks.

5. Translation of Statecharts

There is also great interest in translating the UML/Statemate statecharts directly into Promela, the input language for the model checker SPIN[4]. Promela is a process language, based on the interleaving model of computation. It is a C-like language extended with non-deterministic and loop guarded constructs. Constructs to run processes and for inter-process communication are also provided. Communication takes place via buffered, finite length channels. A Promela specification is usually called a "Promela model".

Latella[16] presents a translation from a subset of UML Statechart diagrams covering aspects of both concurrent behavior, (sequentialization, parallelism, non-determinism, priority), and state refinement into Promela. They consider only a subset of the UML statechart notations without considering history, action and activity states; events are restricted to signal and call events, without parameters; time and change events, object creation and destruction events and deferred events are not considered as are branch transi-

tions; actions are merely a sequence of events without variables and data. The entry/exit actions (of states) are abstracted away. Other object-oriented features of UML statecharts are not included. The UML statecharts are first converted into a hierarchical automata[19] and then translated into Promela. A proof of correctness is also given for the translation function relative to the stated operational semantics.

An earlier work by Holzmann[17] presents a method to translate Statemate statecharts into Promela. This translation allows linear temporal logic model checking of statecharts. Extended hierarchical automata[18] are used as an intermediate form. Two frameworks are presented for either sequential or parallel code with the sequential code being more efficient to verify. To simplify the process, the work was restricted to a sub-language. In statecharts, a transition can be labeled optionally as an event expression, a condition or an action expression. In their sub-set, transition labels were restricted as: i) only Boolean combinations of predicates are allowed in expressions, and ii) the only effect of taking a transition is the generation of events.

6. Component Interaction Testing

Component-based software development is expected to be an effective and widely used method of creating software. However, a major problem is to ensure that components that were developed separately work properly together. Little research and very few techniques exist for testing component interactions, compared to testing individual components or whole systems.

Formal models of component interactions can simplify testing of component interactions by formally defining testing requirements for interactions, and automatically generating required test cases using model-checking technology. This approach allows more rigorous testing, since formal models and test selection criteria do not require individual interpretation by testers.

Other researchers have proposed the use of model-checking technology to generate test cases from formal models. The technique has been applied to hardware, protocols, and specifications of simple control systems, but has not been successfully applied to more complex models of software components. Previous applications have been necessarily simple as state explosion severely limits the size of models that can be model-checked.

The focus of our research is on component interaction testing of software systems. The goal is to detect subtle interaction errors without duplicating the work performed in unit testing. The main obstacle has been obtaining models that capture information about interactions between components. Unit and system testing models are simply not adequate. Interaction errors cannot be found by observing components in isolation. Interaction errors cannot be found by unit testing, and are very difficult to find by system testing as they involve problems with internal actions.

In general, previous formal testing techniques have relied on standard model-checking tools and algorithms for verification. They have not attempted to create efficient algorithms tailored for test generation. Traditional testing focuses on the internal structure and external requirements of units, and uses models such as control-flow graphs or logic specifications. Interaction testing focuses on communication and synchronization between components. Our underlying model is the labeled transition system (LTS). Our approach uses formal models of the components and test requirements. By combining test requirements for different components, intricate interactions can be tested.

In order to facilitate interaction testing, a formal model of the components' interactions must be created. Most of the formal languages suffer from the problems of complex mathematical notation, lack of support for data types, and/or generation of huge state spaces. Initially, we developed the ObjectState[24] modeling language to show the feasibility of creating formal design models for interaction test generation. ObjectState balances the needs of a friendly notation for object-oriented design, formal state-space semantics for test selection, and an efficient resulting state space for test generation.

ObjectState combines architecture description features of UML for Real-Time with a Pascal-like language for modeling component data. In addition, it has a formal semantics that allows it to be reused for interaction testing. ObjectState's behavioral language is a hierarchical FSM. Transitions between states are triggered by receiving messages on ports and/or conditions on local variables. Transitions can perform complex actions on local variables or send and receive messages on ports. States can contain other states hierarchically and can define enter and exit actions. An-

notations describe actions and their effect on local data.

7. REFERENCES

[1] W. Visser, K. Havelund, G. Brat and S. Park, "Model Checking Programs", Proceedings of the *15th International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000

[2] J. Hatcliff, M. Dryer and H. Zheng, "Slicing Software for Model Construction", Journal of Higher-order and Symbolic Computation, 13(4), A special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.

[3]J. Corbett, M. Dryer, et al., "Bandera : Extracting Finite-state Models from Java Source Code", November, 1999. Proceedings of the Internation Conference on Software Engineering (ICSE 2000), IEEE Press.

[4] G. J. Holzmann, "The Model Checker SPIN", *IEEE Transactions on Software Engineering, 23(5)*, pp. 279-294, May 1997.

[5] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[6] R. Valle-Rai, et al., "Soot – A Java Optimization Framework", in *Proceedings of CASCON'99*, Nov. 1999.

[7] J. Hartmann, C. Imoberdorf and Michael Meisinger, "UML-Based Integration Testing", *ISSTA'00*, Portland, Oregon.

[8] J. Offutt and A. Abdurzik, "Generating Test Cases from UML Specifications", *Proceedings of 2nd International Conference on UML'99*, Fort Collins, CO., Oct. 1999, pp. 416-429.

[9] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", in *proc. Of The 3rd International Conference on the Unified Modeling Language (UML)*, York, UK, Oct. 2000, pp.383-395.

[10] H. Yoon, B. Choi and J. Jeon, "A UML-based Test Model for Component Integration Test", *Workshop on Software Architecture and component(WSAC)*, Japan, December 1999, pp.63-70.

[11] Paul C. Jorgensen, *Software Testing – A Craftsman's Approach*, Addison-Wesley, 1995.

[12] Y.G. Kim, H.S. Hong, D.H. Bae and S.D. Cha, "Test cases generation from UML state diagrams", in *IEE Proc.-Softw.*, Vol. 146, No. 4, August 1999, pp. 187-192.

[13] W. Visser, S. Park and J. Penix, "Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking", *FMSP '00*, Portland, Oregon, pp. 3-12.

[14] K. McMillan, *Symbolic Model Checking*, Kluwer, 1993.

[15] S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", *ACM Trans. On Computer Systems*, 15(4), November 1997, pp. 391-411.

[16] D. Latella, I. Majzik and M. Massink, "Automatic Verifcation of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker", *Formal Aspects of Computing* (1999) 11: pp.637-664.

[17] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann, "Implementing Statecharts in Promela/SPIN", in proceedings *WIFT'98 Workshop*.

[18] E. Mikk, Y. Lakhnech and M. Siegel, "Hierarchical automata as a model for statecharts", in *Asian Computing Science Conference (ASIAN'97)*, vol. 1345 of *LNCS*, Springer Verlag, Dec. 1997.

[19] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of UML statechart diagrams", In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, Kluwer Academic Publishers, 1999, pp. 331-347.

[20] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs", ", in Proc. *7th SPIN Workshop*, Stanford University, California, August 30 - September 1, 2000, pp. 245-264.

[21] P. Godefroid, "Model Checking Programming Languages using VeriSoft", in Proc. *24th ACM Symposium on Principles of Programming Languages*, Paris, Jan. 1997, pp. 174-186.

[22] D. Stoller, "Model-Checking Multi-Threaded Distributed Java Programs", in Proc. *7th SPIN Workshop*, Stanford University, California, August 30 - September 1, 2000, pp. 224-244.

[23] D. Park, U. Stern and D. Dill, "Java Model Checking", in *Proc. First International Workshop on Automated Analysis, Testing and Verification*, Limerick, Ireland, June 2000.

[24] W. Liu and P. Dasiewicz, "Component Interaction Testing Using Model Checking", in proceedings of *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2001)*, Toronto, Ontario, June 2001.