Test Generation using Model Checking

Hung Tran
931470260
Prof Marsha Chechik

Abstract

Testing in the software industry is, in general an ad hoc task. There are guidelines to follow but in most cases do not cover sufficient portions of the software product. Recent work has been done to automatically generate test cases such that designers are no longer responsible for designing the test cases but ensuring that the specification of the software is valid. These formal specifications are then used as inputs into the automatic test generation tools, the results of the tool would be a set of test cases that do a better job at covering the range of tests than current non-automated methodologies.

In this paper, we surveay a few such techniques that use model checking technologyas the test generation engine. There are two areas of interest we intend to cover. First is the actual test generation. We discuss a couple techniques that alter the specification to force the model checker to output counter-examples that are then used as test cases for the software application.

Second we examine a few methods that attack automated generation from a state space perspective. That is the specifications that are designed for practical industrial products typically contain far too many states for the model checker to verify to completion. These methods attempt to reduce the state space while maintaining sufficient details that test generation is still possible.

1 Introduction

The testing phase of many software engineering process is generally the final part of the software project; and since being first to market and having a large set of functionality seems to carrys more weight than high quality software, testing is taking on less importance. In turn some errors in the program may not be uncovered until it reaches the customer. This is not to say that testing isn't preformed, but to say that testing isn't performed as rigorously as it should be. Inherent in this is that the test cases used to test these software applications are designed by the developers, who traditionally do not have a good understanding of the product as a whole.

Out of this little discussion there are two areas in testing that should be addressed: Running the test cases and designing the test bed that are run. A solution to runing the tests would be to automate the testing process – as with the build process. This is a pretty trivial problem, most software companies automate their testing in some shape or form, and may additionally have a team that is dedicated to testing the product.

The second area that was addressed is designing the test cases. It would be desireable to automate this process as well, thus removing the human error factor. Thus, we would like tools that can automatically generate test cases, but to do this generation we would need something to generate from, either the program code or a specification. Most test generation tools though use formal specifications because they would have been completed at the beginning of the software cycle whereas the program code is constanly changing; therefore would not be good to base off.

The use of formal specifications in the software development process has many advantages. Firstly the meaning of the specification is precise and unambiguous. Secondly, we may use mathematical proofs to rigorously demonstrate aspects of the specification. Formal specifications have been used for test generation for non-object-oriented software, and lately for object–oriented systems. Yet some of the chief drawbacks to applying formal methods are the difficulty of conducting formal analysis and the perceived or actual payoff in project budget. Testing is a substantial part of the software budget, and formal methods offer an opportunity to significantly reduce testing costs, thereby making formal methods more attractive form the budget perspective.

Automated software test generation also greatly reduces the costs of software testing. Deriving tests from specifications is particularly useful for bodies developing software standards, e.g., W3C, OMG, IEEE, and

ANSI. The goal of these groups is specification development, while the goal of associated companies is software implementation.

Test generation tools though, are not as mature in the software industry as other development tools. There are several reasons for this: Completely automated test generation methods fail mainly due to complexity. Even for small examples, they generate a large amount of test cases that cannot be handled in practice. Other problems are related to the fact that specification based test generation is not applicable equally in all stages of the software cycle. Some of these tools utilize model checkers for their ability to generate counter examples. Model checkers though, have had some success when applied to protocols and hardware designs, but there are fewer success stories with software design.

The focus of this paper is to survey the current research in using model checking to generation test cases. This will touch on areas such as the methodology of properly testing software, the use of model checking to generate tests suits and specialization of specification to suit the needs of test generation. We will also look at how this can be utilized for object oriented software since the software industry trand has been towards object oriented design of is applications. Part of the areas we must look at with model checking is test selection. Since this is an automated process, a large number of test cases can be generated, we must devise also how to properly select the test cases such that they can correctly validate the system using minimal test cases.
We will examine research in both these areas individually

2 Testing Methodology

The overall goal of testing is to provide confidence in the correctness of a program. The only way to guarantee a program's correctness is to execute it on all possible inputs, but this is usually impossible. The most feasible alternative then, is to build a test set that has enough significance to find the maximum number of errors, so that a test that passes gives confidence to the programmer that the program meets its specification, thus correctness.

In general such test sets come in two flavours, specification-based or black box testing and program-based or white box testing. The black box test method (sometimes called behavioral testing) is an approach to find errors in a program by validating its functionalities, without analyzing the details of its code, but by using the specification of the system. The goal is to answer the question: "*Does a program satisfy its specification?*" or, in accordance to the goal of testing, to find if a program does not satisfy its specification. Behavioral tests are much simpler than white box testing because it removes the details of the structure, thus testing at higher levels of abstraction. Additionally, black box testing can test whether a program does not correctly implement incorrect behavior; meaning because of misunderstanding, the program implements a variant of the desired behavior.

White-box testing (also know as structural testing) strategies include designing tests such that every line of source code is executed at least once, or requiring every function to be individually tested. Since behavioural techniques do not use knowledge of the structure of the system, they cannot produce test sets that are as good as structural techniques. For instance there may be parts of the program that are defective because of the implementation or parts of the program may be insensitive to certain inputs. These kinds of problems may not show up in behavioural tests.

In practice, it hasn't proven useful to use a single test design method. A mixture of different methods should be used such that the program isn't hindered by the limitations of a particular test method. Some call this "grey-box" or "translucent-box" test design. Having said that, there are various testing techniques at different levels of abstraction that make use of one or both these testing methods.

At the lowest level of thesting, we have what is called unit testing. Whereby each function, module or class is individually tested. Unit testing has the best ability to control the execution and observe faults of the unit. But, it does not give information about the correct behavior of the system as a whole.

Testing a specific feature together with other newly developed features is know as integration testing. Testing the interface of two components explores how components interact with each other. Integration testing inspects not only the variables passed between two components, but also the global variables. This test phase assumes the components and the objects they manipulate have all passed their local unit tests. With unit testing the only method was structural testing, either method can be used for integration testing.

System testing is designed to reveal defects that cannot be attributed to individual components, or to the interaction among components and other objects. System testing is concerned with issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. In practice, blackbox testing is predominatly used for system testing. The obvious reason is that the number of possible paths that are required to structurally test a program is far too large to handle.

In general formal specifications are defined such taht it is implementation independent. Thus automated test generation using formal specifications cannot easily perform structural testing. This also implies that unit testing isn't easily handled by model checkers. Therefore the focus will be on behavioral testing from a systems level and occasionally an integration level.

2.1 Standard Test Selection Techniques

Having looked at various levels of abstraction of testing we now see how one would like to approach the testing. We could the test the message flows of a system, or states changing, or various other approaches. The following are some of the more frequently used test selection techniques:

2.1.1 Path testing

The most important and widely used testing techniques are based on path testing. This kind of testing uses the flow graph model, which is a graph that captures the flow of processing of the program. The most common kind of flow graphs is derived from the code of a program unit, called control flow graphs. Transaction flow graphs are flow graphs that specify the high-level behaviour of a whole system. Coverage criteria for path testing attempt to cover all paths, all nodes, or all edges.

2.1.2 Dataflow testing

Flow graphs can be annotated with extra information regarding the definition and use of data variables. These are data flow graphs, and the corresponding testing techniques are called dataflow testing. Dataflow criteria attempt to cover all def-use paths, all def-use pairs, all defs, or all uses, among others.

2.1.3 Eventflow testing

Event flows are motivated by data-flow testing. A related step is a step in transition of the design state-machine model with a related event. A prelated path is defined as a path through the model that starts with the first related step, ends with the second related step, and between them there are not other related steps.

2.1.4 State-based testing

State-based testing uses finite state machines (FSM) and various extensions. FSMs typically appear as the specification of the behaviour of systems or objects. They can also be derived from the code with difficulty. FSM models can be tested using path coverage criteria, but there are more sophisticated techniques specifically for them.

2.2 Coverage criteria

How much detail or rigor should be applied to the testing . Stated differently, how large and diverse should the test be be? Based on certain assumptions of the system the rigor of testing can vary.

A coverage criterion is an assumption about how defects are distributed in the program in relation to the program model. In order to obtain reasonable sizes for test sets, rather strong assumptions must be made. These assumptions are based on the following intuitive notions about programs and defects:

- Almost correct programs - programs tend to be almost correct and we only need to worry about a few ways that a program can go wrong.

- Uniform behaviour - programs treat similar inputs similarly. This is a reasonable assumption, but depends on the definition of "similar".

- Errors in special cases - programs tend to work for the general case, but not special cases. This leads, for example, to strategies that test all cases of a program once.

2.2.1 Uniformity assumption

One kind of assumption is the uniformity assumption. We assume that if the program behaves correctly for one input in a class, then it will behave correctly for all inputs in that class. For example, using a control flowgraph model, the statement coverage criteria assumes that if the program passes a test case that exercises a particular statement, then the program will pass all tests that exercise that statement. That is why we assume the program is correct when all statements have been covered by test cases.

2.2.2 Continuity assumption

Another assumption based on the same notion is the continuity assumption: we define a distance metric between inputs, and we assume that if a program is correct for a test case then it will be correct for all test cases that are within a certain distance of that test case.

2.2.3 Regularity assumption

A third kind of assumption based on this notion is the regularity assumption. Here we assume that if the program behaves correctly for all inputs of a certain complexity, then it will behave correctly for all inputs of greater complexity as well. This assumption is based on the fact that programs handle complex inputs by recursively handling simpler inputs. Thus, we may decide to test program with only lists less than 10 elements long, and assume it works for longer lists.

2.2.4 Fault model assumption

The fault model assumption is based on the notion that we only have to test for a limited set of possible errors. We are given a model of the required program and a set of models of possible implementations, some of which are correct, but most are erroneous. We want to make sure that the program does not implement an erroneous model. Using this assumption, for each erroneous model, we select a test case on which it differs from the correct model.

For example, if the requirements are given as a finite state machine (FSM), and we assume that the program implements some FSM with at most *k* states, then we select test cases that distinguish all FSM of at most *k* states that are not equivalent to the required FSM.

2.2.5 Mutation

Another example of a fault model is to apply mutation operators to the required model, and then select test cases that distinguish the correct model from variant models. Usually, we only distinguish mutants obtained by a single mutation because we assume that the implementation is almost correct. This greatly reduces the number of tests required, as well as simplifying test generation.

The first three assumptions do not contain the properties to help i the goal with automation of test generation. However, the fault model and mutation assumptions are a good fit in the model checking paradigm.

3 Testing Object-Oriented Software

In the previous section we discussed testing for the general software program. Testing object oriented programs though are quite different because it is recognized that a component that has been adequately tested in one environment is not always sufficiently tested for any other environment. This implies that we need a detailed investigation of the limitations of the different testing phases: unit testing, integration testing etc. Furthermore, although the parent was deemed correct previously; its methods must be retested when it is subclassed. It remains to be determined what is the best method for doing this. In certain object-oriented languages, multiple inheritances is allowed, making for a very messy test case. [6] suggest that the whole hierarchy tree be flattened and then test the class of interest.

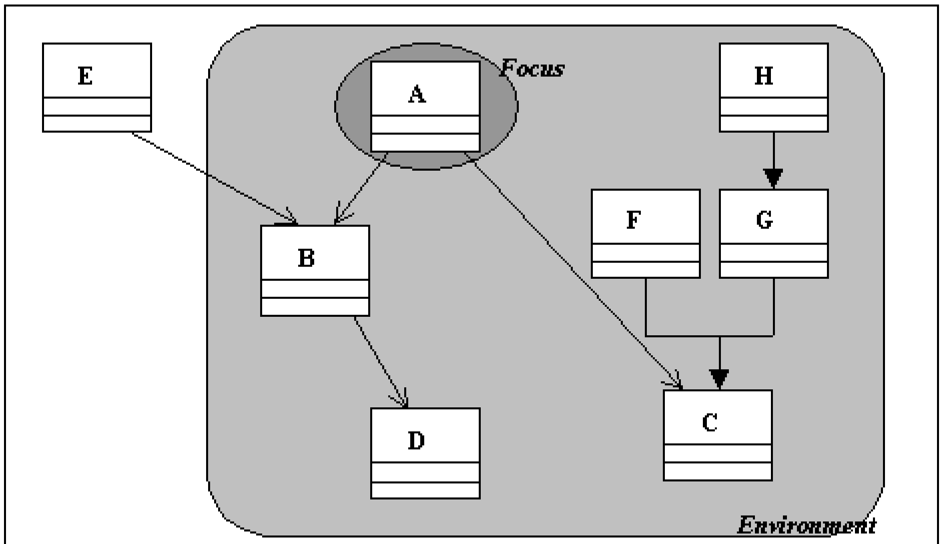

Figure 1. Determining the Focus and the Environment

Additionally, the conventional way of testing software by decomposing into procedures does not hold for the object-oriented world. Part of the reason is that conventional software systems are layered, which made the dependencies easy to trace. In the OO world there isn't a true layering within the software system and therefore the communication paths are not as easily traceable. In the object-oriented paradigm the methods are much smaller thus the inter-method invocations become more abundant, as well as communication between and within classes. Hence the correctness of individual methods within a class can be determined easier, but the interaction between classes and methods are not as easily resolved.
Previously we defined different levels of testing abstractions for conventional software; here we define their equivalent in the object-oriented world.

There are arguments that testing object-oriented software is very similar to testing traditional software, or that it is even simplified, and that it does not require specific strategies. However, testing should take into account the specifics of the object-oriented developments methods, and the structure of object-oriented software. Traditional strategies need adaptation to be fit for object-oriented systems. There have been proposals of testing object-oriented software, but they lack the theoretical basis necessary to ensure the quality of the development, and especially the meaning of a pass case.

As before, there are two methods on how component interaction should be tested: structural and behavioural testing [4]. Structural testing focuses on intra-class interactions. It considers classes as test units, and aims at testing the interactions of the methods of a single object[2]. The levels of granularity coincide with the scope of granularity coincide with the scope of program units, from the compiler point of view. In this approach, the basic units of test are the classes, which are integrated into systems or possibly into subsystem. Behavioural tests is focused on object communication. Behavioural testing considers the meaningful subsequence of system operations as test units, and verifies how the methods of different objects interact to provide a system operation.

There is no basic unit testing for object-oriented software, but there are many kinds of integration unit testing:

- Object integration tests the cooperation of the methods of the object management part of one class with one object of this class. This kind of testing checks that an object satisfies its specification (or, of course, to find counterexamples where it does not) for the object management part. Object integration testing may of course include creators, which are class methods.
- Class integration tests the cooperation of the methods of the class management part of one class with several objects of the same class. The methods of a class are tested in integration to check the class management part of a class. Models from which test data can be derived for object integration and class integration must consider the union of these models.
- Cluster integration tests the collaboration of all methods of several classes with instances of these classes. It is required when objects are too closely tied to be tested individually. Models from which test data can be derived for cluster integration are the same as for object and class integration, but cluster integration must consider the union of these models.

The introduction of stubs should be avoided by introducing a test order that allows the integration of already tested components. This minimizes the testing effort and focuses the test process on successive enrichments of the specification.

The first step is to focus on a particular unit of interest, the focus, that we want to test in detail. This unit can be an object, class or cluster. From figure 2 the focus is A, which uses the units B and C. The unit A can be tested using already tested implementations of B and C or stubs that simulate the behaviour of B and C.

The test environment is the set of all the units visibly used by the focus of interest. This test environment includes all units that are directly and indirectly used. The test environment must also include the subtypes of used units because of the possibility of substitution.

OZTest is a framework for semi-automated generation of object oriented programs that addresses some of these issues. It uses Object-Z as the formal specification language. Object-Z is a model based object-oriented specification language. The system is designed on the premise that the software to be tested is developed in an OO language from an Object-Z specification. The OO language this system uses in this first iteration is Eiffel, potentially other OO languages will be incorporated. One of the phases in OZTest is called configuration flattening, which essentially is flattening of the hierachy structure of each individual classes. OZTest scans each class and stores various attributes in a configuration file. When it comes across an inherited class, it first stores those attributes in memory and for each parent mentioned in the child class, the parent configuration file is loaded, and any renaming of is preformed. Each parent that is loaded has itself already been flattened. Thus all inherited methods are retested in this new context. Test shells[1] are generated from the Object Z specification. For further detail on OZTest see Fletcher and Sajeev [7].

The FREE approach[2] is adapted from a protocol verification technique with formally proven fault finding abilities. This strategy tests the transition of variables from certain sets of values to other sets of values. The abstract states of an object are partitions of the state-space of the object, that is, the set of possible values of the object's variables. The partitioning is based on predicates in the decision statements in the code. A method can cause the object to change its abstract state. These states and transition define the object's state machine. Different state-based testing criteria are used to cover the state-machine.
The idea of the FREE method is to cover the tree of possible transitions from the initial state, without revisiting a state twice. Additionally, it makes the decision to "flatten" the class hierarchy, thus the methods of the parent classes are re-tested in all the derived classes.

The state-based approach extends to testing object interactions. A subsystem is composed of individual communicating state machines. One way of testing the interaction is to regard the subsystem as a large

[1] Test shells are test cases without an oracle to determine the correctness of the results
[2] This method is provided by RBSC Corporation one of their online articles. R.V. Binder; *The FREE Approach to Testing Object-Oriented Software: An Overview*; www.rbsc.com/pages/FREE.html

state machine. This is the approach used in FREE. The same testing criteria can be used for the global state machine. However, due to the complexity of the model, very large numbers of complex tests would be obtained.

4 Model Checkers

Normally, a model checker is used to analyze a finite-state representation of a system for property violations. If the model checker analyzes all reachable states and detects no violations, then the property holds. However, if the model checker finds a reachable state that violates the property, it returns a counterexample – a sequence of reachable states beginning in a valid initial state and ending with the property violation. In this technique the model checker is used as an oracle to compute the expected outputs and the counterexamples it generates are used as test sequences.

4.1 Counterexamples Generation

Next we illustrate [5] that uses a similar approach. To create the counterexamples it uses mutation analysis. [6] also uses model checking to create counterexamples, calling it tweaking of the specification *trap properties*.

4.1.1 Test Generation using mutation Operators

This strategy uses a model checker to generate tests by applying mutation analysis. Mutation analysis is a white-box method for developing a set of test cases which is sensitive to any small syntactic change to the structure of a program. It is a black box test if the test cases are generated from the specification and not from the program code. The rationale is that if a test set can distinguish a program from a slight variation, the test set is exercising that part of the program adequately.

This technique was implemented using SMV.

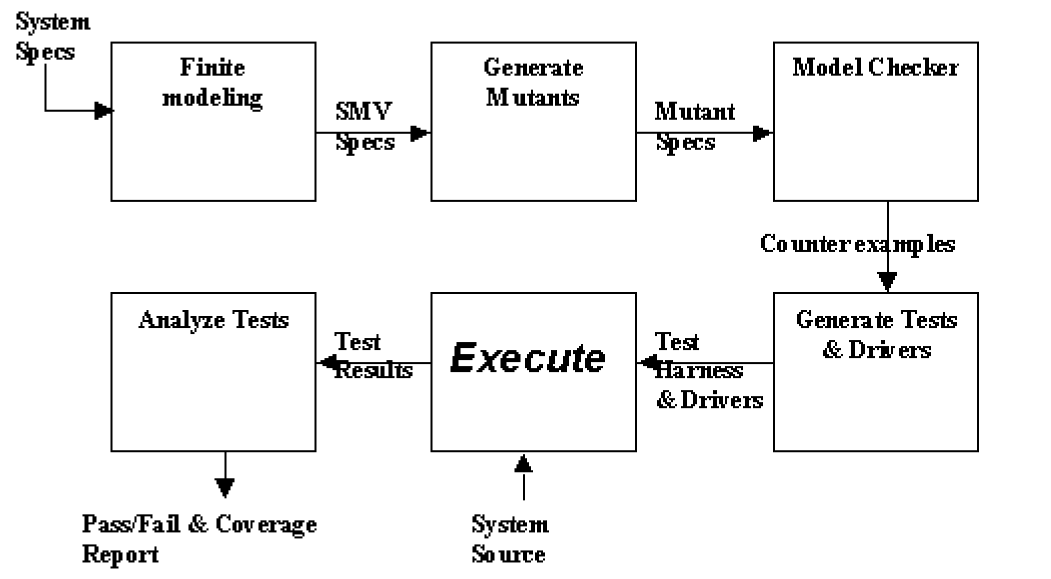

Figure 2. Test Generation using Mutation Operators

The approach of this method is to begin with a system specification and through finite modeling turn it into a specification suitable for a model checker. Mutation operators are applied to the state machine or the constraints yielding a set of mutant specifications. The model checker processes the mutants one at a time. When the model checker finds an inconsistency, it generates a counterexample. The set of counterexamples is reduced by eliminating duplicates and dropping any counterexamples whish is a "prefix" of another, longer counterexample. The counterexamples contain both inputs and expected values and so can automatically be converted to complete test cases. (The test cases generate executable test code, including a test harness and drivers.)

Two categories of mutation operators are identified:

- Changes to the state machine. Since counterexamples comes from the state machine, a good implementation should diverge from corresponding tests. That is, when given the inputs specified by such a test, a correct implementation should respond with different results than those recorded in the counterexample. These are referred to as failing tests – tests that the implementation is expected to fail.
- Changes to the temporal logic constraints. In this category, since the counterexamples comes from the original (correct) state machine, a correct implementation should follow the indicated sequence of states and results. Here we assume a deterministic specification, otherwise the test cases generated from a mutation would not be useful. Thus the implementation is expected to pass these tests.

There are four types of mutation operators used.

1. Addition of a condition to the state machine
2. Deleting a condition where there are multiple conditions
3. Replacing a property substatement with another valid substatement
4. Replacing a property substatement with an invalid substatement

For example given that a transition in the specification:

Next(v):= case c_1: e_1; c_2: e_2; ... esac;

The first type of mutation operators would alter condition c_i by joining some value e to it.
The second type of mutation operators would remove a condition from c_i if there are multiple joined conditions.
Given a CTL property:

SPEC AG $(x{=}mode_i \rightarrow \ldots)$ & $(c_i$ & $x{=}mode_i)$ ...

The third mutation operator would replace $mode_i$ with another valid mode of x.
The forth operator would replace $x{=}mode_i$ with an invalid one, for example $!(x{=}mode_i)$.

Only temporal logic properties that yield counterexamples when violated are modified. Existential assertions that fail are of no interest. For example suppose that the state machine is modified such that an assertion indicating a given state must be reached is no longer true, counter examples for such cases would be difficult to generate. Therefore all existential assertions are removed.

This test code is executed with implementation source which also records coverage. The test code records which results are processed to become a final report of coverage, to show how comprehensive the tests are. This method checks that the implementation fails the cases in which it should fail, and passes when it should pass. Figure 2, shows the flow of this technique. See [5] for a detailed description.

4.2 State Reduction

There are a few techniques that have been used to solve the state explosion problem within model checking. Here we look at a couple techniques that attempt to solve it from the point of view of testing.

4.2.1 Finite Focus

The authors of mutation test generation used a relatively small example to illustrate this procedure. Had an industrial sized model been used an obvious problem would be that the model checker would be unable to handle the state space. This limitation is one of the primary reasons that model checkers have not gained acceptance in the software industry. Finite focus attempts to address this limitation.

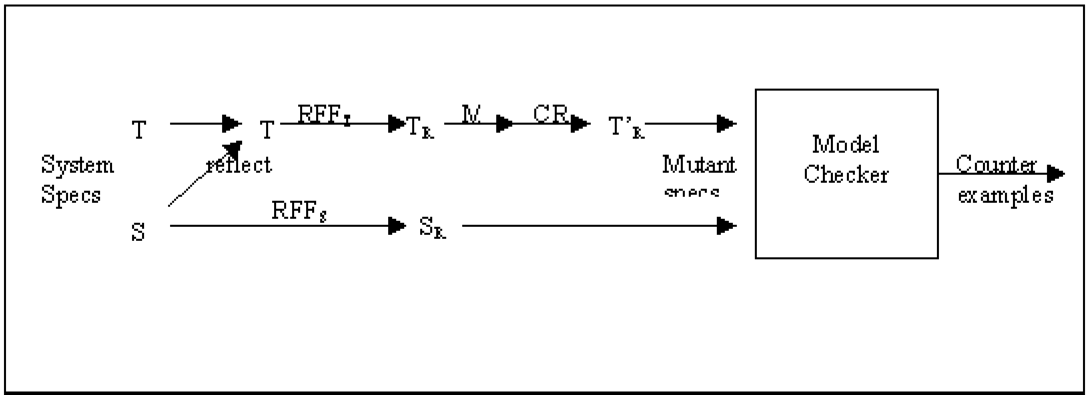

Figure 3. Specification Transformations

Finite focus[10] attempts to preform reduction from the point of view of automated test generation. Thus in traditional model checking where the goal is property analysis and verification, the reductions may summarize states and discard details of transitions. The reduced model may not be precise but it would be useful for the analysis and verification. In terms of automated test generation, in finite focus technique the details may be retained in order to determine if an implementation behaves properly. The tests generated are accumulated from different precise reductions.

A typical abstraction is to map variables with large or unbounded domains to a fixed subset of possible values. From the test generation perspective, the ranges simply need to cover values which may be interesting when used in actual test cases.

For example assume that we are modelling the withdrawal and deposit from a bank account of $1. The area of interest is when the balance in the account is around $1. If this was analysed manually, the focus would naturally be on what happens when the balance is close to zero and ignore, temporarily large balances. Thus amounts greater than $2 would not be of much interest and would be mapped to an “other” state. However, the model checker should know that any set of operations in which the balance exceeds $2 should be ignored. The loss of accuracy in states where the balance is greater $2 is resolved by adding a “soundness” state variable which becomes unsound if the state becomes “other,” . The model checker can then ignore any unsound inconsistencies so that it only returns those which are problems in the full model.

Finite focus views a system specification as a pair of state machine and temporal logic constraints <S,T>. To generate test cases using mutation operators, the reduction for finite focus(RFF) must be in a form that can be analyzed by a model checker. For test generation, the state machine S, is reflected as temporal logic constraints to provide a description for subsequent mutation analysis. Any existing temporal logic constraints, T in the figure, may be added to the reflected constraints which describe the state machine.

Some finite number of states, focused around the initial state, are mapped to states in the reduced specification. All other states are mapped to a single “other” state. The source and destination of each transition are mapped likewise. The function RFF_T maps temporal logic constraints, and RFFs maps the state machine. The two functions, along with constraint rewriting (CR) for soundness, explained below, make up RFF.

RFF_S also adds a separate state machine with the initial state “sound”. Whenever the reduced state machine ends in the “other” state, this added machine goes unsound. It remains unsound thereafter. This step yields a reduced state machine S_R. RFF_T yields reduced temporal logic constraints, T_R, but is less rigidly determined than RFF_S. Together S_R and T_R answers to the finite specifications of the mutation analysis test generator above.

To generate counterexamples, the various mutation operators M are repeatedly applied to the temporal logic constraints. Then, in order to prevent unsound counterexamples the constraints are rewritten such that they are always satisfied when the state is unsound. This constraint rewriting, CR, yields mutated constraints T'_R. Together S_R and T'_R are given to the model checker which computes a number of counterexamples. Soundness for test generation means that any counterexample of the reduced specification (S_R, T'_R) is a valid trace of the original state machine specification, S.

4.2.2 Component Interaction Testing project

The component Interaction Testing method [1] uses formal models of component interactions, and formally defined interaction testing criteria. Formal interaction models are designed by re-using state-machine models developed in object-oriented analysis and design. This technique uses an extension of the Unified Modeling Language, namely ObjectState. The necessary interaction test cases are generated automatically using model checking processes.

ObjectState is comprised of three sub-languages: An architectural description language (ADL) with a representation of components and connections, a behavioral language, with a finite state-machine representation of the behavior of each component, and a data manipulation language, for detailed modeling of the effect of transitions on local component data.

The formal basis for ObjectState models is provided by labeled transition systems (LTS). The LTS formalism models distributed state, interface between components, sequences of interactions, and state-based interactions. It has mature theoretical foundations, efficient algorithms, and many existing tools.

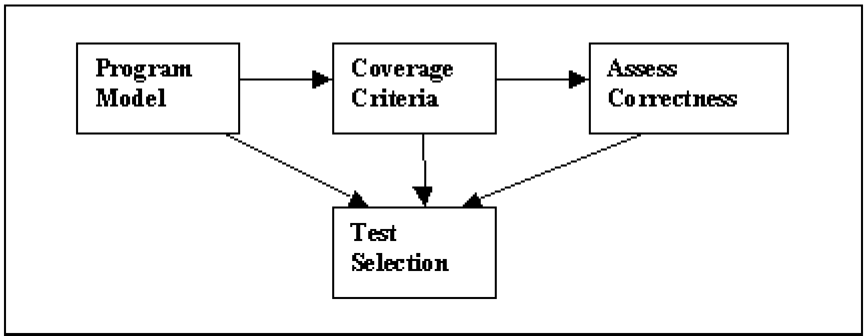

Figure 4. Test Selection technique for Component Interaction Project

Upon defining a formal model, formal coverage criteria can be defined. There is no specifics as to which coverage criteria is chosen, thus it leads us to believe that any of the set of coverage criteria mentioned previously in section 2 can be used. The Component Interaction project uses event flow as the test selection criteria because it exercises user specified pairs of "related interactions".

Two algorithms are provided for test generation with this technique, the first is incremental test generation, which takes advantage of the fact that test cases can be generated incrementally by parts. Partial test cases, with a subset of inputs and outputs are obtained and expanded to obtain a full test case. When considering a subset of inputs and outputs, the other inputs and outputs can be hidden in the system, and the system reduced. Thus to extract, each portion of the test case can require less effort than the entire test case at once. The algorithm selects one of the external ports in the ObjectState model, and hides all messages from the remaining external ports. Components are minimized, and a path to completion is determined.

The inputs and outputs along that path are extracted and a new LTS is created with that sequence of inputs and outputs. This new LTS is a portion of the final test case.

The new LTS is composed with the system, to constrain the behaviour of the system to be compatible with the inputs and outputs selected so far. It reduces the number of possible behaviours of the system and the number of states for the model checker. The algorithm iterates until no external ports remain. The result of the last search is a path with all the exernal inputs and outputs of the system and is hence the final test case.

The second algorithm removes redundant information while preserving all information necessary to generate the test case. For test case generation interactions with the environment are preserved. Interaction between components is not preserved, but only the effect of the interaction with the rest of the system. If two interactions do not move the other component into the same state, but equivalent states, then the two interactions can be merged. Since it is not known which states are equivalent until the interactions in the other component have been merged, the algorithm must proceed iteratively; it optimistically merges interactions initially, but when it processes the other component it may find it made a mistake. Then it re-processes the previous component. The algorithm ends when it finds no more mistakes. The algorithm reduces each process individually in an iterative computation, which avoids state explosion problems.

5 Conclusion

Formal methods, typically used in the specification and analysis phases of software development, offer an opportunity to reduce the cost of the testing phase. We introduced a few techniques that utilize formal methods to provide the backbone of software testing. These techniques used model checking to verify these specifications and provide counter examples to systems that did not meet the specification of the model.

Additionally, model checking traditionally have been used to verify properties of specifications, but the enormous state space of finite-state models of industrial strength sofware specifications often lead to the state explosion problem: the model checker runs out of memory or time before it can analyze the complete state space. This occurs even when partial order and other methods for reducing the state space are applied. Model checking is thus usually more effective in detecting errors and generating counterexamples then in verifcation.

We saw that there is a lot of potential for model checking in automated test generation. However, there is still a lot that needs to be done before model checking techology is full utilized in software testing. There are additional areas of improvement such as reduce cost and complexity and increase ease of use.

7 Reference

[1] W. Lui, P. Dasiewicz; *Component Interaction Testing Using Model-Checking*; Submitted to 5th European Conference on Software Maintenance and Reengineering, CSMR 2001, September 11 2000

[2] H.S. Hong, Y.R. Kwon, S.D. Cha; *Testing of Object-Oriented Programs Based on Finite State Machines*; Proceedings of Asia-Pacific Software Engineering Conference '95, Brisbane, Australia, pp 234-241, Dec 1995

[3] S. Barbey, D. Buchs, C. Péraire; *A Theory of Specification-Based Testing for Object-Oriented Software*; Proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy, October 1996, LNCS (Lecture Notes in Computer Science) 1150, EPFL-DI-LGL, 1996, pp. 303-320

[4] S. Barbey, D. Buchs, C. Péraire, A Strohmeier; *Incremental Test Selection for Specification-Based Unit Testing of Object-Oriented Software Based on Formal Specification*; Technical Report EPFL-DI No 99/303, Swiss Federal Institute of Technology in Lausanne, CH-1015 Lausanne. Switzerland, 1998

[5] P.E. Ammann, P.E. Black, W. Majurski; *Using Model Checking to Generate Test from Specifications*; Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia (December 1998), edited by John Staples, Michael G. Hinchey, and Shaoying Liu, IEEE Computer Society, pages 46-54.

[6] A. Gargantini, C. Heitmeyer; *Using Model Checking to Generate Test from Requirements Specifications*; Proc., Joint 7th European Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symposium on Foundations of Software Eng. (ESEC/FSE99), Toulouse, FR, Sept. 6-10, 1999

[7] R. Fletcher, A.S.M. Sajeev; *A Framework for Testing Object Oriented Software Using Formal Specifications*; In Alfred Strohmeier, editor, Reliable Software Technologies (Ada-Europe '96), volume 1008 of Lecture Notes in Computer Science, pages 159-170. Montreux, Switzerland 1996. Springer-Verlag 1996.

[8] W. Liu, P. Dasiewicz; *Formal Test Requirements for Component Interactions*; Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'99)

[9] W. Liu and P. Dasiewicz; *Selecting System Test Cases for Object-oriented Programs Using Event-Flow*; Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'97)

[10] P. Ammann, P.E. Black*; Abstracting Formal Specifications to Generate Software Tests via Model Checking*; Proceedings of the 18th Digital Avionics Systems Conference (DASC), St. Louis, Missouri (October 1999), IEEE, volume 2, pages 10.A.6.1-10. NIST-IR 6405 (extended version), 1999

[11] R.J. Allen, D. Garlan, J. Ivers; *Formal Modeling and Analysis of the HLA Component Integration Standard*; Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 1998