Knowledge-based Software Test Generation

Valeh H. Nasser, Weichang Du, Dawn MacIsaac
Faculty of Computer Science, University of New Brunswick
Fredericton, NB, Canada
{valeh.h, wdu, dmac}@unb.ca

Abstract

Enriching test oracles with a test expert's mental model of error prone aspects of software, and granting control to them to specify custom coverage criteria and arbitrary test cases, can potentially improve the quality of automatically generated test suites. This paper reports our investigation on the application of knowledge engineering techniques in automated software testing to increase the the control of test experts on test generation; ontologies and rules are used to specify what needs to be tested and reasoning is used for identification of test objectives, for which test cases are generated. An architecture of the ontology-based approach to testing is presented and a prototype which is implemented for unit testing is described with a case study.

1. Introduction

Granting control to a test expert to utilize their knowledge about error-prone aspects of software and specify what needs to be tested, can result in generation of a smaller yet error-revealing test suite. At the current stage of the research we are focusing on model-based testing, but the proposed knowledge based approach can be generalized and applied to white-box and black-box software testing.

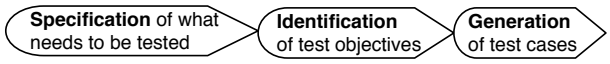

Figure 1. Three Concerns of Test Generation

There are three concerns in test generation (Figure 1): *specification* of what needs to be tested, *identification* of Test Objectives (TOs) based on the specification, and *generation* of test cases for the identified TOs. These three aspects can be tightly coupled.

The specification of what needs to be tested is addressed by definition of *test oracle*s and *Coverage Criteria* (CC), which specify the correct behavior of the software and requirements on the generated test suite [21] respectively. This aspect of software testing is crucial, because it determines the quality of the generated test suite. Zhu *et* al. [21] categorize CC as *structural*, which specify what elements of software should be tested (such as All Transition Pair coverage [14]), *fault-based*, which uses some measurement of fault-detecting ability of the test suite (such as mutation-based methods [16]), and *error-based*, which is based on error-prone aspects of software (such as boundary testing [12]).

The second aspect of software testing is identification of test-objectives based on the specification of what needs to be tested. A test-objective delineates a single test case, and is identified with an algorithm. This concern can be tightly coupled with the first concern, because the identification algorithm can be tightly coupled with the specification of what needs to be tested. There are several approaches to identification of TOs: Explicit identification of TOs by a test expert [11]; the use of identification algorithms with the rules implicitly built into them [14]; provision of a language for defining CC rules and use of identification algorithms that rely on the specification language [8]; and translating CC into temporal logic and have a model checker to identify TOs [18].

The third concern in automated test generation is generation of test cases for the identified TOs. The test cases are generated based on a test oracle with several approaches: One approach is to use graph traversal algorithms [14, 4]. Another approach is using model-checking tools for test-case generation [19]. With this approach it is asserted that there is no path with the required specification in the model. The model checker tries to find the required path and returns it as an output. A third approach is using AI planners to generate test-cases [16]. AI Planners are used to generate paths

to reach identified goals.

Test oracles and CC are used for specification of what needs to e tested. The knowledge that is conveyed by test oracles can be at different levels of abstraction (i.e. code, design, or requirements [15]). An issue with abstraction is that poor abstraction can be a barrier to generating high quality test suites [5], if it removes the knowledge essential for specification of error-prone test cases. To solve this, Benz [5] demonstrates how abstraction of error-prone aspects of software can enhance test-case generation by defining system-specific CC. The error-prone aspects, which are also used in Risk Based Testing [1] are software elements that are more likely to produce an error and can be domain specific (such as concurrent methods, database replications, network connection), from general test guidelines and experience (such as boundary values), or system specific and revealed in interactions of testers with developers and designers (such as use of an unreliable library). Granting control to test experts to extend a test oracle with knowledge about error-prone aspects of the system and specify custom CC rules can increase the control of the test-expert on the automated test generation and, therefore, potentially enhance the quality of the generated test-suite.

To exploit a test expert's knowledge in automated test generation, the first two concerns of test case generation, i.e. specification of what needs to be tested and identification of TOs need to be decoupled. The decoupling makes the test oracle extensible and enables it to support specification of arbitrary test cases, implementation knowledge, invariants on model elements, distinguished states [17], and knowledge about error-prone aspect of the system. Also the system should support standard CC which are accepted in literature, as well as additional CC rules which are based on test experts' mental model.

In this work knowledge engineering techniques is used to decouple the specification of what needs to be tested and the identification algorithm of TOs and, therefore, enable a test expert to enrich the test oracle and specify custom CC accordingly. To achieve this, the test oracle and expert knowledge (EK) are represented in ontologies, which are connected together. The common CC rules and expert defined CC rules are used to specify what needs to be tested. Reasoning is used to identify the TOs. Then the test cases are generated for the identified TOs using model checking, AI planning, or graph traversal algorithms.

The rest of this paper is organized as follows. Section 2 describes how knowledge based approach is used to deal with the three aspects of software testing. Section 3 describes an implementation for unit testing based on the UML State Machines (SMs). Section 4 illustrates a concrete example of unit test generation and specification of arbitrary expert knowledge. Section 5 concludes the paper.

2. Ontology based Approach

Our proposed ontology based approach to software test generation focuses on separation of the three concerns of test case generation as follows.

Specification of What Needs To Be Tested The knowledge that is used in specification of what needs to be tested is externalized in ontologies and rules. An extensible ontology based test oracle is connected to an expert's mental model. The mental model ontology is expert defined and can include knowledge about implementation, error-prone aspects, etc. Based on the vocabulary defined by the ontologies, CC rules, which are either standard or expert defined, are specified. CC rules are in the form shown below. The TO selection criteria specify a condition that should hold on some elements for them to be a part of structure of a test case.

```
TO :- TO selection criteria
```

The TOs are specified using the structural properties of corresponding test cases, which *directly* or *indirectly* make them candidates as TOs. For instance, in unit testing based on the UML SMs, a TO can specify that transition tr_1 of the SM should be traversed immediately after transition tr_2 is traversed. This TO can be directly required because every possible sequence of two transitions is required to be covered. This sequence can be indirectly required because the two transitions have a *definition-use* relationship [20], which is required to be tested.

Identification of TOs Reasoning on the test oracle and CC rules is used to identify test-objectives. Also, before a test case is generated for an identified TO, it is determined whether a test-case that satisfies the TO already exists in the test suite. This is done by reasoning on the partially-generated test suite, which is represented in an ontology. This approach reduces the number of redundant test cases. To this end, redundancy checking rules for a given TO need to be generated.

Generation of Test Cases A test-case for a given TO is generated using a test-case generation method such as AI planning, graph traversal, model checking,

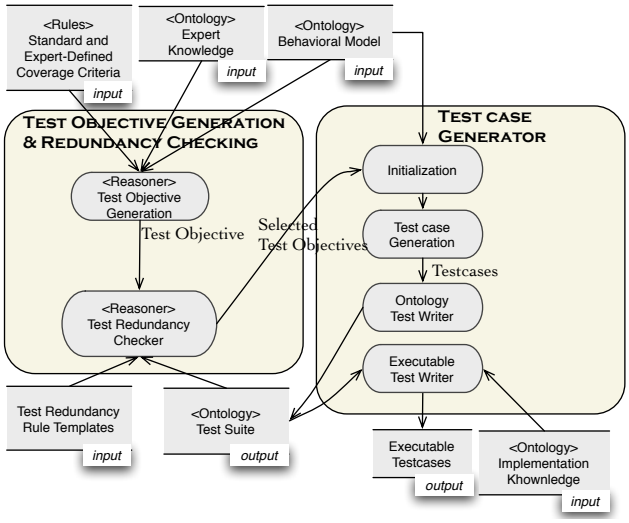

Figure 2. Ontology-based Test Generator

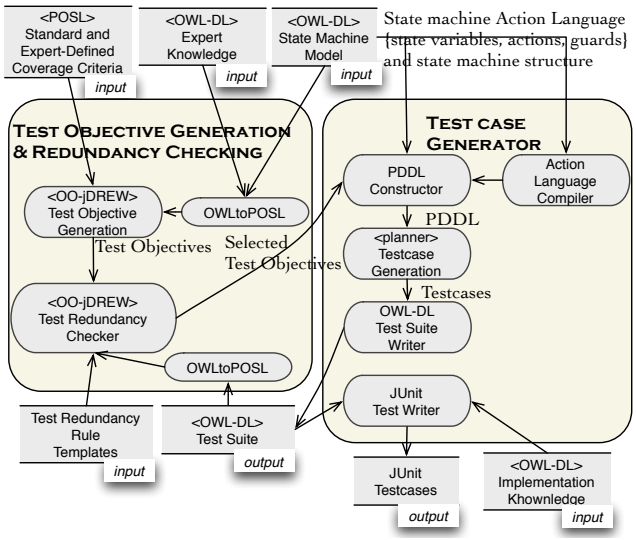

Figure 3. Implementation Technologies

etc. and written in a programming language independent test-suite ontology. The executable test cases are then generated from the test suite ontology, using a programming language dependent implementation knowledge ontology. The implementation knowledge ontology, which can be reverse engineered from the source code and connected to the model ontology, conveys information such as name of implemented methods, state variable setters and getters, etc.

Figure 2 shows the architecture of the system which has two main processes: *TO Generation and Redundancy Checking* and *Test Case Generator*.

The TO Generation and Redundancy Checking is implemented using two reasoners. One reasoner is used to generate TOs from test oracle ontology which is connected to expert knowledge ontology, and CC rules. The other reasoner uses the redundancy checking rules and the test suite ontology which is generated so far to check whether a given test-objective is already satisfied in the test-suite or not. If it is not, the TO is accepted, otherwise it is discarded. If the TO is accepted, then a test case is generated for it and added to the test suite ontology. Then the redundancy checking reasoner continues to select another test-objective. The Test Case Generator is in charge of generating test cases for the selected TOs. The generated test cases are added to the test suite ontology, which is used to reduce redundant test-cases. The initialization subprocess, initializes the input of test case generation process. Finally the executable test cases are generated from the test suite ontology.

An ontology based representation of test oracle, and rule based specification of CC, promotes separation of specification of what needs to be tested and identification of TOs, and, therefore, form a flexible mechanism for specification of various CC. The system empowers the test expert to use knowledge about the system to control the test-suite by specifying CC.

3. Implementation

The architecture that is discussed in the previous section is abstract and can be realized using various technologies. This section describes an implementation of the system for *SM based unit testing* and the technologies which are used to realize the system, including OWL-DL, POSL, and OO jDREW, and an AI planner named Metric-FF [10] (See Figure 3).

The UML SM, expert knowledge, implementation knowledge, and test suite are represented in OWL-DL [3]. A TBox ontology specifies different concepts and relations in them and an ABox ontology specifies a particular instances of them by importing the TBox ontology and instantiating the elements in it. The TBoxes are reusable while the A-Boxes are for an instance system. The Ontology Definition Metamodel (ODM), which is adopted by the OMG, has a section that describes the UML 2.0 metamodel in OWL-DL. However, a prototype ontology which is much simpler and less modifiable is used for the purpose of this work. The XMI [13] representation of the SM can be converted to the ontology-based representation automatically. The implementation knowledge can be automatically imported if the source code of the unit is available. Some

parts of the ontologies are visualized in next section.

The TO Generation and Redundancy Checking component use OO jDREW [2] for reasoning tasks. The OWL-DL Ontologies are transformed into POSL [6] using the mappings presented by Grosof *et* al. [9]. The CC and the Test Redundancy Rule Templates are written and converted to POSL respectively. Examples of CC and Redundancy Checking Rule Templates are provided in the next section. The TOs are specified with *structure predicates* which specify some structural properties of the test case. Ideally a TO should specify why it is selected and its structure, to be used by the redundancy checking reasoner and the Test Case Generator process respectively. A set of structure predicates, which are used to specify the structural properties of the test cases are used to compose CC and TOs. Examples of TOs are provided in the next section.

The Test-case Generator process, uses an AI planner called Metric-FF [10] for test-case generation. The inputs to Metric-FF are the problem and domain description in the PDDL 2.1 language [7], which are provided by the planner initializer. The compiler-subprocess is in charge of parsing the state variables, guards and actions of the SM which are specified in an SM action language. The inputs of the planner is initialized based on data from the SM and structure predicates of a TO. The generated test cases, include methods to be called at each step, their inputs and the expected values of the state variables. The generated test cases are then given to the Test Suite Writer sub-process to be written back to the Test-Suite Ontology in OWL from which the JUnit test cases are generated.

4. Case Study

Figure 4 visualizes the SM of a cross road traffic light controller. The traffic light stays green for at least 'long time interval' on one direction and turns yellow when a car is sensed in the other direction. Then it remains yellow for 'short time interval' before it becomes red. There is a correspondance between the SM elements and class under test: The state variables correspond to the state variables of the class; The events correspond to the methods; The actions simulate how the state variables are changed by the methods. The traffic light SM does not have a timer and delegates the counting responsibility to another class which produces call events.

Some parts of the ontological representation which is converted to POSL is visualized in Figure 5. A CC in POSL for All Transition Pair coverage and the query that is asked from OO jDREW to generate TOs are as follows:

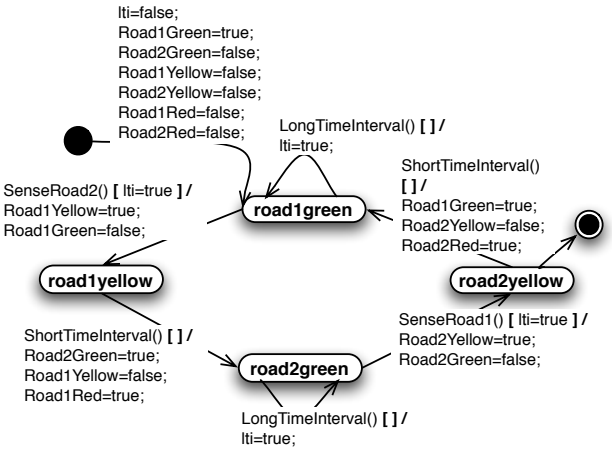

Figure 4. Traffic Light SM

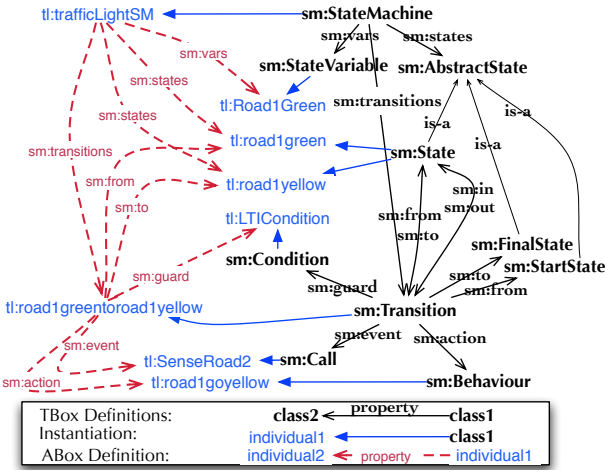

Figure 5. SM Ontology

```
Rule: coverage([immediate],[?a,?b]):-transition(?a),
transition(?b),notEqual(?a,?b),from(?a,?state),to(?b,?state).
Query: coverage (?predicates, ?args).
```

Before generating a test case for a TO, a redundancy checking rule is generated for it; test suite ontology (see Figure 6) is translated into POSL; and OO JDREW is used to check whether the TO is already satisfied. A TO and the corresponding redundancy checking rule are shown below:

```
coverage([immediate],[start2road1gr,road1grtoroad1gr]).
Redundancy Rule: exist(?t,?st1,?st2):-
hascall(?st1,start2road1gr),hascall(?st2,road1grtoroad1gr),
hasstep(?t,?st1),hasstep(?t, ?st2),nextstep( ?st1, ?st2 ).
```

For an unsatisfied TO, AI planning is used to generate test cases which are added to test suite ontology ABox. To generate executable JUnit test cases, implementation knowledge ontology is used (Figure 7).

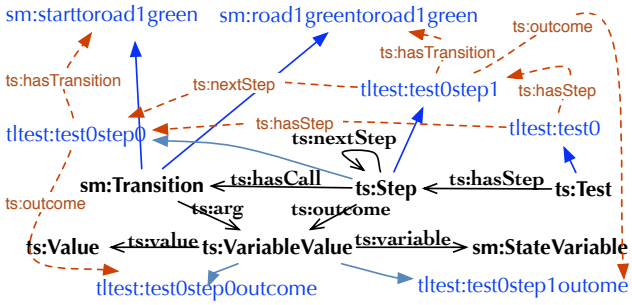

Figure 6. Test suite Ontology

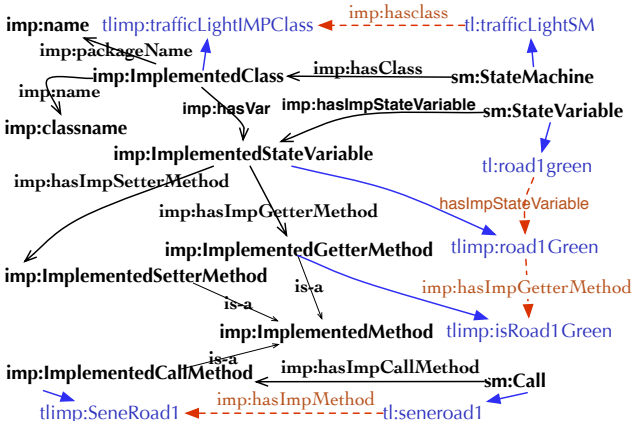

Figure 7. Implementation Knowledge

4.1. Other Coverage Ciriteria

While some CC such as All Transition Pair coverage merely depend on the structure of a SM, some other CC refer to additional expert knowledge which can be expressed in an ontology and be used to define custom CC. As an example, a test expert's knowledge about use of an unreliable library can be modeled, and CC for a test suite that tests every call to the library once can be specified. This knowledge can be attached to the unit's SM ontology (Figure 8). A CC can be defined as it is shown below. The test structure indicated by this CC is that at a transition t, the state variable sv, must have value val.

```
coverage([AtTransitionStateVariableHasValue], [?t,?sv,?val]):-
untestedlib(?l),method(?m), belongsto(?m,?l),behaviour(?b),
uses(?b,?m),transition(?t),action(?t,?a),
variablevalue(?vv), risky (?m,?vv), value(?val),
statevariable (?sv),hasvar(?vv,?sv), hasvalue(?vv,?val).
```

Another example is a CC that defines if a state variable has a boundary value in a state, then a transition that has a behavior that uses the value should be tested

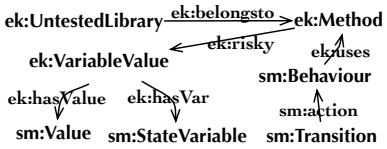

Figure 8. Use of an Unreliable Library

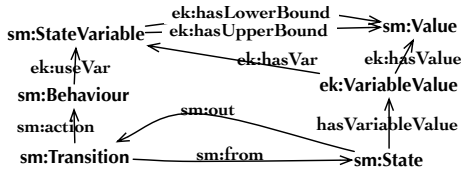

Figure 9. Boundary Values

[12]. Figure 9 visualizes the expert knowledge and the CC can be as follows:

```
coverage([AtTransitionStateVariableHasValue], [?t,?sv,?val]) :-
state(?s), vaiablevalue(?vv), hasvar(?vv,?sv),
hasvalue(?vv,?val), hasboundary(?sv,?val),transition (?t),
from (?t,?s),behaviour(?b), action(?t,?b), usevariable(?b,?sv).
hasboundary(?sv,?val):-lowerbound(?sv,?val).
hasboundary(?sv,?val):-upperbound(?sv,?val).
```

Other CC can be implemented by representing the knowledge in an ontology and defining rules that refer to vocabulary defined by the ontology. For instance, to implement all content dependance relationship coverage [20], which requires that every use of a variable be sequenced after every definition of the variable in a test case, the definition-use relationships among the behaviors and guards can be added to the ontology. Another example is Faulty Transition Pair coverage [4], which required error states, the transitions to them be modeled in an ontology. Then a rule that generate objectives to go to error states are generated. Implementing Full Predicate coverage[14] required more effort.

5. Concluding Remarks

This work studies the use of knowledge engineering techniques in software testing. The benefit that knowledge engineering techniques can bring to automated testing is specification of extensible test oracles which can model test experts' mental model and lend themselves to definition of custom CC. In unit testing, a unit can be large and central to the system. The method grants control to a test expert to specify what test cases should be generated and therefore potentially increases a test suite's quality. Other benefits of the system are that the generated test suite ontology is programming

language independent and can be translated into different languages, and the expert knowledge TBox ontology is re-usable.

Providing for enriching the test oracle with additional knowledge, enables blending white-box and black box testing into gray box testing. While rigid test oracles and CC hinder a test expert in using their knowledge to control automated testing, an extensible CC allows them to add any knowledge to the test oracle and define CC accordingly. Either arbitrary test cases can be defined or rules can be used to generate TOs. Rules can be defined based on a model's structural elements and/or generally accepted, domain-specific, and/or system specific error prone aspects of software.

Although the system is extensible, manipulating the CC and knowledge required knowledge engineering skills. Further research into test experts' mental model, and a method of presenting the system to a test expert abstractly, so that it can be easily learnt is required. Also, the CC which are accepted by literature should be implemented in the system. Further research needs to be done into how test experts describe test cases and a language of structure predicates needs to be devised accordingly, for the test expert to specify the structure of test cases using TOs. This work concentrated on unit testing and the next step is using knowledge engineering in integration testing and system testing.

References

[1] J. Bach. Risk-based Testing. *Software Testing and Quality Engineering Magazine*, 1(6), 1999.

[2] M. Ball. OO jDREW: Design and Implementation of a Reasoning Engine for the Semantic Web. Technical report, Technical report, Faculty of Computer Science, University of New Brunswick, 2005.

[3] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, L. Stein, et al. OWL Web Ontology Language Reference. *W3C Recommendation*, 10, 2004.

[4] F. Belli and A. Hollmann. Test generation and minimization with" basic" statecharts. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 718–723, 2008.

[5] S. Benz. Combining test case generation for component and integration testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 23–33, 2007.

[6] H. Boley. POSL: An Integrated Positional-Slotted Language for Semantic Web Knowledge. http://www.ruleml.org/submission/ruleml-shortation.html, 2004.

[7] M. Fox and D. Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(2003):61–124, 2003.

[8] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 134–143, 2002.

[9] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 48–57, 2003.

[10] J. Homann. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[11] A. Howe, A. Mayrhauser, and R. Mraz. Test Case Generation as an AI Planning Problem. *Automated Software Engineering*, 4(1):77–106, 1997.

[12] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 139–150, 2004.

[13] Object Management Group. XML Metadata Interchange (XMI) specification. http://www.omg.org/technology/documents/formal/xmi.htm, 2007.

[14] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML'99 - The Unified Modeling Language. Beyond the Standard.* Springer, 1999.

[15] T. Ostrand and M. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[16] A. Paradkar. Plannable Test Selection Criteria for FSMs Extracted From Operational Specifications. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 173–184, 2004.

[17] A. Paradkar. A quest for appropriate software fault models: Case studies on fault detection effectiveness of model-based test generation techniques. *Information and Software Technology*, 48(10):949–959, 2006.

[18] S. Rayadurgam and M. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[19] S. Rayadurgam and M. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In *Eighth IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, page 83, 2001.

[20] Y. Wu, M. Chen, and J. Offutt. UML-Based Integration Testing for Component-Based Software. In *Cots-Based Software Systems: Second International Conference*, pages 251–260, 2003.

[21] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.