An Ontology-based Software Test Generation Framework

Valeh H. Nasser and Weichang Du and Dawn Maclsaac

Department of Computer Science, University of Calgary, Calgary, AB, Canada
Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada
E-mail: v.hnasser@ucalgary.ca, {wdu, dmac}@unb.ca

Abstract

In automated test generation, granting control to test experts over test selection can enhance quality of generated test suites. However, in many cases test experts’ control is limited and they can not define custom coverage criteria. This work proposes a general framework for application of knowledge engineering to software testing, which facilitates specification of custom coverage criteria and provides means for generating test cases from different artifacts in dissimilar domains.

1. Introduction

In automated test generation a test suite is automatically generated based on a software artifact (e.g. code, design diagrams, or requirements). A crucial issue in automated test generation is test selection, which is about what tests should be included in the test suite for a desired quality to be achieved [11]. One approach to address this issue is allowing test experts select test cases manually. While this approach may be required in some cases, it is not efficient [6]. A second approach is automatically selecting test cases to satisfy some predefined coverage criteria [9]. With this method, test experts do not have much control over the test suite that is being generated. A third method which can lend more control to test experts over the generated test suite, is to leverage specification of custom coverage criteria. The test cases are then generated based on the coverage criteria specifications, which can be domain or system specific [7].

Benz [4] demonstrates that utilization of domain-specific coverage criteria which are based on error-prone aspects of software can enhance the quality of a generated test suite. According to Bach [2] error-prone aspects, also used in Risk Based Testing [2], are software elements that are likely to produce an error and can be: (1) domain specific (such as concurrent methods, database replications, network connection), (2) based on general test guidelines and experience (such as boundary values), or (3) system specific and revealed in interactions of testers with developers and designers (such as use of an unreliable library). In addition to utilization of test experts mental model about error-prone aspect of the system, Paradkar [10] suggests that a test case generation method is required that supports specification of arbitrary test cases, implementation knowledge, invariants on model elements, and distinguished states.

However, in many cases test case selection algorithms are tightly coupled with coverage criteria and software artifacts (such as [9]). This inhibits definition of custom coverage criteria for a specific domain or system.

In order to increase control of test experts over test selection, in the past few years, we have been extensively involved in the design of testing techniques that benefit from knowledge engineering techniques. For instance, we have previously proposed a knowledge-based system architecture that generates unit tests from UML state machines [7]. This system externalizes knowledge about what needs to be tested in the form of ontologies and rules based on which reasoning algorithms are used for the generation of high-level description of tests. Also, an ontology based description of the generated test suite is maintained in order to enable test redundancy checking using standard reasoning mechanisms. In the current paper, we have tried to capture the experiences that we have gained in the design of knowledge-based testing techniques in a generalized ontology-based test case generation framework. Our proposed general framework provides means for generating test cases from different artifacts in dissimilar domains.

The rest of this paper is organized as follows. Section 2 provides an overview of the framework. Section 3 and 4, respectively describe specifications that are used in the method and phases of test generation from the
specifications. Section 5 further discusses the framework. Section 6 concludes the paper.

2. Framework Overview

The ontology based test case generation framework uses knowledge engineering techniques to decouple test selection algorithms from the knowledge that is used for test selection. The decoupling allows test experts to freely manipulate the ontology based representations of specification of what needs to be tested, without the need to modify hardcoded test selection algorithms. To do this, ontologies and rules are used to specify what needs to be tested, and reasoning algorithms are used to select high level description of test cases, which are called test objectives. The selected test objectives are then translated into abstract test cases. The abstract test cases are represented in an ontology, which allows the use of reasoning for redundancy checking. Finally, executable test cases are generated from the abstract test case ontology.

Hence, the executable test suite is generated in four phases, which are depicted in Figure 1 and described below:

**Phase 1- Test Objective Generation:** The ontology based representation of what needs to be tested which includes: behavioural model specifications, expert knowledge, and coverage criteria rules, is used to generate a set of test objectives. This phase is conducted using reasoning on the ontologies, which describe the specifications.

**Phase 2- Redundancy Checking:** For every test objective, a redundancy checking rule is used to check whether the test objective is satisfied by a test case, which is already included in an abstract test suite ontology. This phase is also conducted using reasoning on the ontologies.

**Phase 3- Abstract Test Suite Ontology Generation:** For each non-redundant test objective, an abstract test case is generated and added to the partially generated abstract test suite ontology. This phase is run using prevalent test generation methods from literature (such as graph traversal algorithms [9]).

**Phase 4- Executable Test Suite Generation:** For every abstract test case in the abstract test suite ontology, an executable test case is generated using the implementation knowledge ontology.

Phase 1 generates a set of test objectives. After phase 1 is completed, phase 2 and phase 3 are performed repeatedly to generate an abstract test suite. Then in phase 4, based on the abstract test suite, the executable test suite is generated.

3. Specifications

With this framework, in each phase some specifications are transformed into another form (Figure 1). This section describes the specifications which are used in the framework.

**Behavioural Model Ontology** The behavioural model ontology is an ontology-based representation of the software artifact based on which tests are generated. This ontology describes concepts corresponding to the software artifacts structural elements, the relationships between them, and their instances.

For example, for UML artifacts, this ontology can be created based on the the Ontology Definition Meta Model (ODM) [1] standard by the OMG, and be automatically populated from the XML Metadata Interchange (XMI) [8] representation of existing models. For code-based artifacts, this ontology can describe the concepts in a programming language (such as methods,
function call, etc.) and be automatically populated from the existing code. For GUI testing this ontology can describe the GUI elements and be automatically populated from the existing GUIs.

**Expert Knowledge Ontology** The expert knowledge ontology extends the behavioural model ontology with knowledge that is beyond the behavioural model ontology. This knowledge is exploited by the coverage criteria rules for identification of test objectives. This ontology describes test experts' mental model and is an extension point that facilitates support of various coverage criteria rules. The ontology has the advantage of retaining this knowledge, which is used by test experts.

This ontology can be designed based on error taxonomies, bug databases, and commonly used coverage criteria. Examples of pieces of knowledge that can be included in this ontology are: knowledge about use of an unreliable library, boundary values of state variables or inputs, expected exceptions, variable definition and use, concurrency relationships, and user interaction points.

**Test Objectives** A test objective delineates a test-case and provides a language for test experts to define the test cases abstractly and decouples test case selection from test case generation. It consists of two parts: the predicateList and the parameterList, which are lists of predicates and parameters separated by a comma. A predicate in the predicateList has one or more parameters, which are listed respectively in the parameterList.

[The list of predicates separated by comma]
[The list of parameters separated by comma]

A test objective specifies a condition that must hold on some elements in a corresponding test case. The predicates specify the conditions. The parameters specify the elements which are defined in the behavioural model ontology or their values. A test objective specifies the structural properties of a test case. A test case can be described by combining test objectives.

Examples of test objectives for test case generation based on UML state machines are as listed below, where $Trans1$, $Trans2$, $Var1$, and $Val1$ are respectively a transition, a transition, a state variable, and a value defined in the behavioural model ontology:

[CoverTransition][Trans1], [Immediate][Trans1, Trans2], [After][Trans1, Trans2], [AtTransitionStateVariableHasValue][Trans1, Var1, Val1]

Similarly for code-based testing and GUI testing test objectives can be designed as listed below:

[CoverMethodWithInputs][Method1, InputValues1]

**Coverage Criteria Rules** The coverage criteria rules are test case selection rules and can be expert-defined, system/domain-specific, or standard. In general, a rule follows the form below. The :- symbol denotes a deduction from the right side to the left side of the rule.

test objective :- test objective selection criteria.

The test objective selection criteria describe the conditions that should hold on some elements for them to be a part of the structure of a test case. The test objective specifies the structure of selected test cases. Coverage criteria rules refer to the vocabulary defined by the behavioural model and an expert knowledge ontology. The body of a rule specifies conditions on the parameters of the test objective, which is described in the head of a rule.

An example of a coverage criterion is shown below. In the examples, a variable name starts with a question mark.

Conditions:
1- A variable's domain has some boundaries,
2- A method uses the variable,
3- The variable's value is a user input.
Test Objective:
- Check if the method throws an OutOfBoundaryException.

coverage([ThrowException], [?Method, ?Variable, ?Value,OutOfBoundaryException]) :-
StateVariableBoundaryValue (?Variable, ?Value),
Uses (?Method, ?Variable)
UserInput(?Variable).

In this example the behavioural model ontology, which can be reverse engineered from the source code, includes the knowledge that defines methods and variables and the Uses relationships between them. The boundaries of the variables and whether their value is a user input is described in the expert knowledge ontology. The expert knowledge ontology can be created manually or automatically from formal documents such as annotations made by programmers in a source code. Based on these knowledge bases and the coverage criteria, reasoning can be used to generate test objectives.

**Abstract Test Suite Ontology** The abstract test suite ontology, which is linked to the behavioural model ontology, describes the test suite. Abstract means that it can be implementation-neutral and programming-language-neutral, depending merely on the knowledge provided by a software artifact. The abstract test suite ontology consists of a set of abstract test cases. An abstract test case is specified by a list of steps, input
values at each step, and values of state variables after each step. A step corresponds to an event that changes the state of the system. The ontology-based representation of the test suite allows use of reasoning to identify redundant test cases.

Redundancy Checking Rule Templates

Redundancy checking rule templates are used to generate redundancy checking rules for test objectives. A redundancy checking rule facilitates checking whether a test objective is already satisfied by a test case, which is included in a test suite. The redundancy checking rules follow the form below:

The test objective is satisfied by the test suite :-
the structural characteristics of a test case that satisfies the test objective.

The structural characteristics of a test case that satisfies the test objective are described using the vocabulary defined by the abstract test suite ontology and the behavioural model ontology. For every test objective, a redundancy checking rule is generated based on redundancy checking rule template. For every test objective predicate, there exists a redundancy checking rule template.

Implementation Knowledge Ontology

The implementation knowledge ontology is used for translating the abstract test suite ontology to an executable test suite. This ontology extends the behavioural model ontology with implementation information. The knowledge represented by this ontology, which is programming-language-dependent, can include: variable getters and setters, implementation names of methods, classes, namespaces, constructors, etc. This ontology can be automatically populated, if the source code is available. This ontology helps postponing the task of generation of actual executable test cases to after the implementation is done. Also it enables maintaining the same set of abstract test cases while the implementation details change.

Executable Test Suite

An executable test suite is the main output of the system, and is the result of the abstract test suite being translated using the implementation knowledge. This test suite is written in a programming language for an implementation of the system under test. If the API of an automated testing framework such as JUnit [5] is used, the tests can be executed and verified automatically.

4. Phases by Example

This section describes the phases of test generation from the specifications using a simple partial example of unit testing based on UML state machines. Only relevant parts of the example ontologies are shown in the figures. During the phase 1, which is test objective generation, initial test case selection is conducted using reasoning and the output, which is a very high level test suite is presented as a set of test objectives. Figure 2 illustrates the example which generates test cases based on a UML state machine. In this example test objectives that specify tests that pass transition t1 and transition t2 of the state machine are generated.

In phase 2, which is redundancy checking, for every test objective, reasoning is used to checks a partially generated abstract test suite ontology for existence of a test case that satisfies the test objectives. Figure 3 illustrates an example, in which the test test1 of the abstract test suite ontology satisfies the test objective, which specifies a tests that passes transition t1. Hence,
5. Discussion

The objective of this method is provision of a framework that can be extended by specification of custom coverage criteria rules. In order to extend the system, one can specify the corresponding coverage criteria rules in a logic programming language and populate the referred knowledge to the expert knowledge ontologies. Hence, this method is as powerful as the rules and ontologies that are specified in it.

The method can be exploited to generate test cases based on various software artifacts such as code, design, requirements, etc. For this purpose, a behavioural model ontology representing the software artifact is used and the artifact is populated into the ontology.

Additionally, the method can be applied to different domains, such as GUI testing, concurrency testing, networking etc. To apply the method to different domains, the knowledge about error prone aspects of the domain is added to the domain-specific expert knowledge ontologies. New coverage criteria are defined based on the vocabulary defined in the behavioural model and expert knowledge ontologies. Also, the test cases can be manually specified by manually specifying test objectives.

Furthermore, rules can be defined to support standard coverage criteria from literature. Example rules for several standard state machine based coverage criteria [9, 3] are listed below:

- **All Transition Coverage**: coverage([?covertransition],[?t2]) :- transition (?t2).
- **All Transition Pair Coverage**: coverage([?immediate],[?t1, ?t2]) :- transition (?t1), transition(?t2, notEqual(?t1,?t2), from(?t1,?state), to(?t2,?state)).
- **Faulty Transition Pair**: coverage([?immediate],[?t1, ?t2]) :- transition (?t1), transition(?t2), faulty(?t2), notEqual(?t1,?t2), from(?t1,?state), to(?t2,?state).

Redundancy checking rules for the corresponding test objectives are as follows:

- **[covertransitions], [TR]**:
  
  test(?t), hasStep(?t,?st1), hasTransition(?st1,?t).
  
  - **[immediate], [TR1,TR2]**:
  
    exist() :-
    test(?t), hasStep(?t,?st1), hasTransition(?st1,?t),
    hasStep(?t,?st2), hasTransition(?st2,?t),
    next(?st1, ?st2).

However, as described below, the method has some limitations:

**Non-Optimal Redundancy Checking** The redundancy checking process is not optimal. This is because the redundancy checking works backwards. That
is, it only checks whether a test case that covers the current test objective already exists in the test suite. It does not guarantee that a test objective will not be satisfied by a test case generated later. Thus the method does not necessarily generate a test suite of optimal size. The order of selection of test objectives affects the size of the generated suite.

**Time Efficiency** For larger units under test, test case generation can be slow because of the use of reasoners. The reasoning algorithms are not fast and this is the trade-off between efficiency and generality. For large systems the method may not operate interactively.

**Modelling Complexity** If the method is used with a model-based test generation methodology, the complexity of modelling software artifact is a challenge. However in presence of UML models, the behavioural model ontology can be automatically populated from the XMI representation of the existing model. Similarly, to reduce the complexity of creating expert knowledge, it may be possible to provide tools that populate it from existing documents that convey the knowledge in another format, such as source code or custom annotations made by programmers within source code. Likewise, to deal with the complexity of creating implementation knowledge ontology, the ontology can be populated by reverse engineering existing source code.

**The Limitation of Logic Programming** Writing rules can be complicated and the expressiveness of the rule languages are limited due to the limitations of the reasoning algorithms. The constraints on the expressiveness of the rule languages limit the coverage criteria rules and redundancy checking rules that can be supported by the system. For instance, in Horn logic no universal identifier can be in the body of the rules. Hence, rules like the example below can not be expressed:

\[ \text{Predicate2}(x) \text{ :- for all x, Predicate1}(x) \]

This limitation can be overcome for finite domains by listing all possible values of x in a list and recursively iterating over the list to check the values of Predicate1 with them. But such an ‘extensional’ approach can not be very efficient for large domains.

6. **Concluding Remarks**

This work presents a general knowledge based test case generation framework that allows defining custom domain-specific and/or system-specific coverage criteria for various software artifacts and domains. By using custom coverage criteria, test experts can control what tests are included in generated test suites. For this purpose, the framework uses reasoning on ontologies to address the test case selection issue.

However, further research should be done before the framework can be applied in practice. A test objective language, which is used for high level specification of test cases should be developed. Ontologies should be developed for the software artifacts and domains under test. Defining domain-specific coverage criteria and expert knowledge ontologies is not a trivial task and should be conducted in research labs. Additionally further research should be conducted on integration of the framework with existing development methodologies.

**References**


