A Model-Based Regression Test Selection Technique

Leila Naslavsky Hadar Ziv Debra J. Richardson

Donald Bren School of Information and Computer Sciences
University of California, Irvine

{lnaslavs, ziv, djr}@ics.uci.edu

Abstract

Throughout their life cycle, software artifacts are modified, and selective regression testing is used to identify the negative impact of modifications. Code-based regression test selection retests test cases sub-set that traverse code modifications. It uses recovered relationships between code parts and test cases that traverse them to locate test cases for retest when code is modified. Broad adoption of model-centric development has created opportunities for software testing. It enabled driving testing processes at higher abstraction levels and demonstrating code to model compliance by means of Model-Based Testing (MBT). Models also evolve, so an important activity of MBT is selective regression testing. It selects test cases for retest based on model modification, so it relies on relationships between model elements and test cases that traverse those elements to locate test cases for retest.

We contribute an approach and prototype that during test case generation creates fine-grained traceability relationships between model elements and test cases, which are used to support model-based regression test selection.

1. Introduction

As software systems grow in size and complexity, so does the need for higher-level models and abstractions in their development. As a result, model-centric development has been widely adopted; it uses structural and behavioral models as guidance for coding. This adoption can also be attributed to the dissemination of the Unified Modeling Language (UML) [6]. [ref?]

Model-centric development creates opportunities to drive testing processes at higher abstraction levels and to demonstrate code to model compliance, through Model-Based Testing (MBT). MBT employs models to generate test cases based on what is expected of the system. MBT tests what the code is supposed to do, thus complementing code-based testing, which tests what the code does. Another advantage of MBT is support for certain testing activities when code is unavailable. For example, missing code obstructs activities that often rely on information available through code analysis, such as regression test selection.

Regression test selection identifies the negative impact of modifications applied to software artifacts throughout their life cycle. In traditional approaches, code is modified directly, so code-based selective regression testing is used to identify negative impact of modifications [8, 13]. In model-centric approaches, modifications are first done to models, rather than to code. Consequently, the negative impact to software quality should be identified by means of selective ***model-based regression testing***. To date, most automated MBT approaches focus primarily on automating test generation, execution, and evaluation, while support for model-based regression test selection is limited [12].

Approaches to model-based selective regression testing select test cases if they traverse modified model parts. They require relationships between model elements and corresponding test cases traversing those elements. Usually, relationships are obtained as an afterthought: analyzing potential test cases model coverage, or simulating models with abstract test cases. We contribute a different perspective by creating these relationships during the test case generation process.

Our approach adapts traditional selective regression testing techniques to support model-based regression test selection. It uses model transformations to support test case generation and to address traceability through creation of relationships among transformed models [10]. When models are modified, these relationships are used to locate abstract test cases for re-run. This paper describes the approach and results obtained through a case study.

2. Approach and Prototype

Our approach adopts UML class and sequence diagrams as its modeling perspective. It generates test cases from models while creating an infrastructure comprised of test-related models and fine-grained relationships among these models. This infrastructure is used, in turn, to support the identification of test cases for retest. Figure 1 below describes the process.

The first phase creates the models and traceability infrastructure: (1) transforms sequence diagrams into model-based control flow graphs (hereafter called *mbcfg*) and saves traceability model; (2) transforms *mbcfg* into a test generation hierarchy, saves traceability model, and complements test generation hierarchy with empty slots for completion with abstract test case input data values. Input data values are systematically created using a constraint solver (e.g. the USE tool [7]). Abstract test cases are further transformed into concrete test script skeletons.

The second phase uses the models and the infrastructure from the first phase, as well as the modified UML models: (1) compares two models to create a differencing model; (2) transforms sequence diagrams into modified *mbcfg*; (3) uses differencing model and traceability models to do a pairwise graph traversal of original and modified *mbcfg*; (4) use selected dangerous entities identified during (3) to classify test cases.

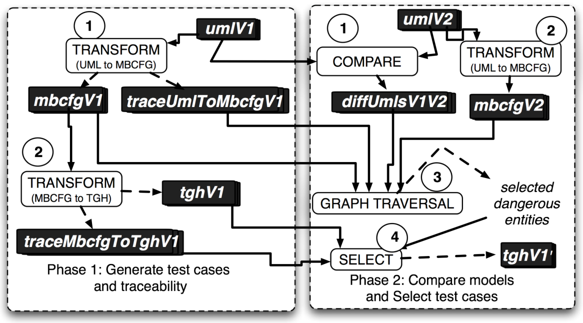

Figure 1. Approach Overview

The prototype uses the Eclipse IDE, *ATL* and *EMF Compare* plug-ins [4, 10]. Test-related models comprise the infrastructure created during the first phase. Each model instantiates a metamodel described below (more in [11]).

UML: Class and sequence diagrams describe structural and behavioral views, respectively. Sequence diagrams model messages flow between objects. The approach creates abstract test cases comprised of calls to boundary classes, which play the role of intermediating interactions between actors (or test drivers) and the system.

Model-Based Control Flow Graph: These models are alternate views to sequence diagrams' that are used to support abstract test cases generation. The approach derives paths that achieve particular graph coverage (e.g. path coverage) and identifies inputs that cause paths to be traversed [1]. This metamodel is a variation of an inter-procedural restricted control flow graph (IRCFG) [14], extended with loop constructs.

Test Case Generation Hierarchy: We employ a testing template framework concept to organize generated test cases [15]. It describes valid input data space partitions hierarchically based on testing strategies. In our approach, roots represent sequence diagrams' input spaces. Derived paths are stored under each corresponding root. They represent expected behavior for test cases.

Traceability: Model-centric solutions create relationships among transformed artifacts during the transformation process to support traceability [10]. We build upon the Atlas Model Weaver (AMW), which is a model weaving solution with a core weaving metamodel that was extended to support traceability [5]. Our approach uses it to externalize fine-grained relationships among test-related models.

Differencing: We adopt the *EMF Compare* plug-in for model comparison [4]. It compares EMF models and creates match and differencing models. The differencing model stores information on changes required to transform original into modified models (e.g. *add*, *remove*, and etc.).

2.1 Types of Modifications Considered

Our approach identifies class and sequence diagrams' modifications. Modified entities in one view might impact entities in the same or different view, so our approach relies on impact analysis to detect affected entities. Indirect model modifications are included into the differencing model during model comparison.

Related works on (UML) model-based selective regression testing rely on taxonomy for UML diagrams' changes [2]. Our approach identifies similar direct changes, but it also uses *mbcfg* information to support impact analysis on behavioral models (section 2.2). The following are examples of direct class diagram changes and how they would impact other entities: (1) If a **class attribute** that comprise an OCL constraint (e.g. operation pre-, pos-condition) is changed, the OCL constraint is considered changed; (2) If an OCL constraint navigates a changed **association**, that OCL constraint is considered changed; (3) if a **class invariant** is changed, all operations of the class are considered changed (including the constructor).

2.2 Indirect Change Impact on Behavioral Model

Our approach selects test cases to re-test the *implementation*. Thus, the change impact identification on behavioral models aims at locating entities *in the model* that might require *implementation* modification. It seizes existence of *mbcfg* along with the traceability models to perform necessary impact analysis.

Nodes in *mbcfg* represent message calls to operations on class diagrams (figure 2). Nodes are organized hierarchically: *implementation* of operations called with container nodes (message) includes calls to operations called with contained nodes. For example, *implementation* of operation *op_1* in *B* (called with *m1*) should have two alternative branches: one calls operations *op_2* and *op_3*, the other calls operation *op_4*.

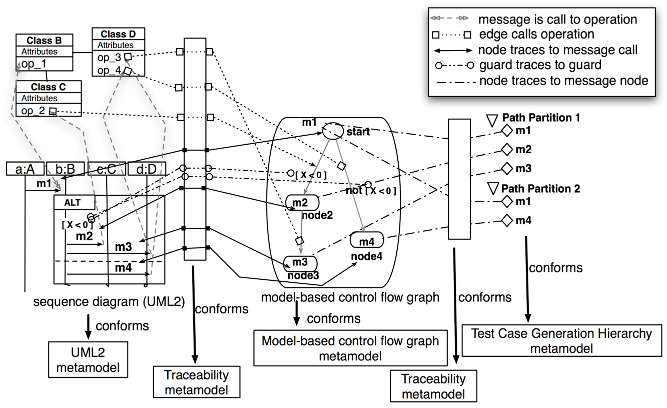

Figure 2. Traceability between UML model and MBCFG.

Changes to *op_2*, *op_3*, or *op_4* **could** impact *op_1*'s *implementation*. However, the sequence diagram shows that messages *m2* and *m4* are only sent under certain conditions. If operation *op_2* were changed, only one branch would be impacted, so *op_1* would not be considered impacted. On the other hand, operation *op_1* could be called with a different message (i.e. same operation is called with a message other than *m1*, possibly in different sequence diagram). It means there is no unified expected behavioral description for operation *op_1*. As a result, one cannot assume changes to *op_2* only impacts *m1-m2* branch. Therefore, *op_1* would be considered fully impacted because its expected behavior might be modified. The impact analysis evaluates all modified (in class diagram) operations to identify how such change might impact operations that call them. It creates a *list of impacted operations* used during graph traversal.

2.3 Graph Traversal

We adapt the algorithm in [8] to perform traversal of *mbcfg* (figure 1, phase 2). The original algorithm performs control flow graphs' pairwise walk looking for modified statements. When modified statements are found, incoming edges are added to a *set of dangerous edges*.

The adapted algorithm checks if an edge leading up to a node was modified, prior to checking for node modifications. The edge is considered modified if it has a modified constraint (guard). Guards' modifications are identified using traceability relationships to locate corresponding guards in the UML model (figure 2, arrow named "guard trace to guard"). Modified edges are added to the *set of dangerous edges*. Identification of modified guards results in addition of all other edges with the same tail to the *set of dangerous edges*. Indeed, a guard change might result in modified test cases' expected behavior.

Nodes' equivalence is identified using traceability relationships to locate the corresponding operations in the UML model (figure 2, arrows "node traces to message call", and "message is call to operation"). Then, it checks if that element was modified looking it up in the differencing model and in the *list of impacted operations*. Node modification also results in addition of triggering edge to the *set of dangerous edges*.

2.4 Test Case Selection

Abstract test cases derived from models describe expected result and expected behavior. For instance, sequences of message calls in sequence diagrams describe expected behavior. Such test cases are fully explored when expected result and behaviors are used as oracle. Thus, abstract test cases consist of an identifier *n*, an input *i*, the expected model execution trace $Mb_{ET}(i)$, and the expected result $Mb_{R}(i)$. Input *i* abstractly represents test script drivers (triggering message, and/or the test case arguments).

Code-based regression test selection techniques assume specification immutability, while model-based techniques select abstract test cases based on **models'** modifications. Thus, it is mandatory to re-visit definitions of obsolete, reusable and retestable.

Obsolete test case: Test cases are obsolete if their input *i* had been modified, despite changes to $Mb_{ET}(i)$ and $Mb_{R}(i)$. It includes direct changes to originating sequence diagrams' pre-conditions when they constrain test case values domains.

Restestable test case: Test cases are retestable if they are non-obsolete (model-based) test case and they traverse modified model elements. Retestable might require modifications to oracles ($Mb_{ET}(i)$ and/or $Mb_{R}(i)$) codified into a concrete test script.

Reusable test case: Reusable test cases are test cases from the original test suite that are not obsolete or retestable. Hence, these test cases do not need to be re-executed.

Traceability models are used to support classification of abstract test cases into obsolete, retestable and reusable. Given a *set of dangerous edges* in *mbcfg*, traceability relationships (figure 2, arrow named "node traces to message node") are used to locate paths in the test generation hierarchy (figure 2, "path partition 1" and "path partition 2") related to dangerous edges in *mbcfg*.

We assume there is a correspondence between abstract test cases and concrete test scripts, so corresponding concrete test scripts can be located for re-run given selected abstract test cases.

3. Example

We compared our approach to others with regards to efficiency, precision and safety [13], using the Personal Investment Management System, which is a system to track investments [9]. We created 1 class diagram and 17 sequence diagrams based on use cases, design documents and code. We modified the code to isolate interface, logic, and database layers, which allowed us to focus on testing of actual functionalities.

Our tool generated 56 abstract test cases that were transformed into JUnit skeletons. We used the UML-based Specification Environment tool and it's extended Snapshot language to solve path constraints and to systematically obtain test case values [7]. The code was instrumented, and tested with completed JUnit test cases. Upon model changes, test cases were selected using: our tool and manual application of a representative model-based regression test selection technique (RTStool) [2]. Code was also changed to accommodate model changes (obeying our regression bias), and test cases were selected with a representative safe code-based tool (DejaVOO) [8].

Changes were selected based on the number of *impacted* elements [3]. We performed two sets of changes (v.2 and v.3). Set v.2 consists of one change of each type in the taxonomy [2]. It comprises 8 logical changes totaling 45 direct changes to class and sequence diagrams, which impacted 6 class diagram elements, 9 guards and selected 8 dangerous edges. Our approach selected 12 test cases for retest, RTStool 19 and DejaVOO 8. Set v.3 aimed at obtaining a higher number of *impacted* elements. It comprises 2 direct class diagram changes, which impacted 34 class diagram elements, 20 guards and selected 11 dangerous edges. Our approach selected 16 test cases, RTStool 22, and DejaVOO 16. With both change sets, it selected subsets of RTStool's selected sets, and equal or superset of DejaVOO's selected sets. This indicates it can achieve higher precision, while not sacrificing safety.

There are some threats to the validity of obtained results. First, we created the models from code and specification documents constrained by UML and OCL constructs acceptable by our prototype solution. Our approach's impact analysis performs advantageously over RTStool when major functionality operations are unified in a single sequence diagram. If all modified operations were called with more than one message, our approach would not achieve better precision. Other threats to validity include possible errors in our manually applying the RTStool.

4. Conclusions and Future Work

We presented an integrated solution to model-based test generation and regression test selection. It entails traceability relationships required to support regression test selection are created *during test case generation*. We compared our approach to other approaches using a case study and the regression test evaluation framework [13].

Our approach assumes models are primary artifacts in the software development process. Code generation tools should evolve and become reliable to the point that testing compliance of code with models at the adopted abstraction level will be unnecessary. Future work includes addressing use of models at higher abstraction levels and support for test generation based on other modeling perspectives and strategies.

5. References

[1] Binder, R. V., Round-trip Scenario Test, Testing Object-Oriented Systems - Models, Patterns, and Tools, Addison-Wesley, 2000, pp. 579-591.

[2] Briand, L. C., Labiche, Y. and Soccar, G., Automating Impact Analysis and Regression Test Selection Base on UML Designs, TR Carleton University, 2002.

[3] Briand, L., Labiche, Y., O'Sullivan, L., Sowka, M., Automated Impact Analysis of UML Models, Journal of Syst. and Softw. (Elsevier), 79 (2006), pp. 339-352.

[4] Brun, C., Pierantonio, A., Model Differences in the Eclipse Modeling Framework, The European Journal for the Informatics Professional, IX (2008), pp. 29-34.

[5] Didonet Del Fabro, M., Bézivin, J., Valduriez, P., Weaving Models with the Eclipse AMW plugin, *Eclipse Modeling Symposium, Eclipse Summit Europe,* 2006.

[6] Forward, A., C. Lethbridge,T., Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals, *Int. Wrksp. Models in Softw. Eng.,* 2008.

[7] Gogolla, M., Bohling, J., Richters, M., Validating UML and OCL Models in USE by Automatic Snapshot Generation, Journal on Softw. and Syst. Mod., 4 (2005), pp. 386-398.

[8] Harrold, M. J., et al., Regression test selection for Java software, *conference on Object-Oriented Programming, Systems, Languages, and Applications,* 2001.

[9] Jalote, P., An Integrated Approach to Software Engineering, Springer, 2006.

[10] Jouault, F., Loosely Coupled Traceability for ATL, *European Conference on Model Driven Architecture, Workshop on Traceability,* 2005.

[11] Naslavsky, L., Ziv H., Richardson D. J., Towards Traceability of Model-based Testing Artifacts, *3rd Wrksp. on Adv. in Model Based Testing,* 2007.

[12] Naslavsky, L., Ziv, H., Richardson, D. J., Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches, TR ISR, UC Irvine, 2006.

[13] Rothermel, G., Harrold, M. J., Analyzing Regression Test Selection Techniques, IEEE Trans. on Softw. Eng., 22 (1996), pp. 529-551.

[14] Rountev, A., Kagan, S., Sawin, J., Coverage Criteria for Testing of Object Interactions in Sequence Diagrams, Fundamental Approaches to Software Engineering, Springer Berlin / Heidelberg, 2005, pp. 282-297.

[15] Stocks, P., Carrington, D., A Framework for Specification-Based Testing, IEEE Trans. on Softw. Eng., 22 (1996), pp. 777-793.