Information and Software Technology 51 (2009) 16–30

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automating regression test selection based on UML designs

L.C. Briand [b,*], Y. Labiche [a], S. He [a]

[a] Carleton University, Software Quality Engineering Lab, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada
[b] Simula Research Laboratory and University of Oslo, P.O. Box 134, Lysaker, Norway

ARTICLE INFO

Keywords:
Regression testing
Test selection
Object-oriented software engineering
UML

ABSTRACT

This paper presents a methodology and tool to support test selection from regression test suites based on change analysis in object-oriented designs. We assume that designs are represented using the Unified Modeling Language (UML) 2.0 and we propose a formal mapping between design changes and a classification of regression test cases into three categories: Reusable, Retestable, and Obsolete. We provide evidence of the feasibility of the methodology and its usefulness by using our prototype tool on an industrial case study and two student projects.

1. Introduction

The purpose of regression testing is to test a new version of a system so as to verify that existing functionalities have not been affected by new system features [12,19]. Regression test selection is the activity that consists in choosing, from an existing test set, test cases that can and need to be rerun to ensure existing, unmodified functionalities are still working correctly. Reducing the number of regression test cases to execute is an obvious way of reducing the cost associated with regression testing, which is usually substantial [19].

The main objective of selecting test cases that need to be rerun is to identify regression test cases that exercise modified parts of the system. This is referred to as *safe regression testing* [27] as, in the ideal scenario, it identifies all test cases in the original test set that can reveal one or more faults in the modified program. In order to achieve such an objective, we need to classify test cases in an adequate manner. Adapting definitions in [18], we aim to automatically classify test cases as follows:

- *Obsolete*: A test case that cannot be executed on the new version of the system as it is 'invalid' in that context. Classifying a test case as obsolete may lead to either modifying the test case and corresponding test driver or removing the test case from the regression test suite altogether.
- *Retestable*: A test case is still valid but needs to be rerun for the regression testing to be safe.
- *Reusable*: A test case that is still valid but does not need to be rerun to ensure regression testing is safe.

Regression test selection can be based on source code control flow and data flow analysis. In this case, based on information about the code of the two versions of the program, one selects test cases that execute new or modified statements (in the new version of the program) to be rerun, or formerly executed statements that have been deleted from the original version of the program [28]. This selection is based on an analysis of the changes at the source code level to determine their impacts on test cases. A drawback is that it requires that the changes be already implemented but it can be very precise in terms of selecting a minimum regression test set as complete change information is available. (Precision varies among code-based regression test selection strategies [27].) An alternative, and complementary approach, is to use architectural/design information available in design models [31]. In this case, selected test cases execute new or modified model elements (e.g., class operations in the case of a UML model), or model elements formerly executed but deleted from the original version. The impact of possible changes is first assessed on the design of the last version of the system, by comparing what would be the new design with the existing design. The change impact magnitude is then assessed and a change management group decides whether to implement it in the next version of the source code. Assuming there is traceability between the design and regression test cases, we can, at the end of the design impact analysis, automatically determine what regression test cases will need to be rerun and what test cases should be removed from the regression test suite as they are no longer valid. Therefore, one main advantage of a design-based approach is the possibility of performing early regression test planning and effort estimation.

Another motivation for working at the architecture/design level is in part motivated by efficiency as discussed in [12,19]. Leung and White note that the cost of selecting regression test cases to rerun must be lower than the cost of running the remaining test cases for

* Corresponding author.
E-mail addresses: briand@simula.no (L.C. Briand), labiche@sce.carleton.ca (Y. Labiche), siyuan@sce.carleton.ca (S. He).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.09.010

test selection to make sense. In [12], it is suggested that working closer to the architectural level may be more efficient than at the source code level. To summarize, the motivations for investigating test selection techniques at the architectural or design level are fourfold, the last two points being related to efficiency:

- We can estimate the extent of the effort required for regression testing earlier on, at the end of the design of the new system version. Estimating regression test effort is an important part of impact analysis and one of the decision criteria to include a change in an upcoming version (the modification-request problem [12]).
- Regression test tools can be largely programming language independent and they can be based on a standard, widely used design notation such as the UML.
- Traceability between code and test cases requires to store and update dependencies between test cases and code statements or other representations of the code, e.g., control flow graphs. Managing traceability at the design level may be more practical than doing it at the code level as it enables the specification of dependencies between test cases and the system at a higher level of abstraction.
- No complex static and dynamic code analysis is required (e.g., data flow, slices). The latter analysis being usually necessary for identifying possible dynamic bindings between methods at run-time [29]. Using UML designs enables the easy retrieval of relevant static and dynamic information (e.g., class interactions at run-time from sequence diagrams) since they provide information at a higher level of abstraction than the source code.

There are, of course, potential drawbacks too. For example, using designs for impact analysis and test selection requires the designs to be complete, internally consistent, and up-to-date. Though CASE tools are getting better at providing round-trip engineering capabilities, this is not always easy in practice. Another issue is that some (potentially faulty) changes to the source code may not be detectable from UML documents, e.g., a change in a method's body (a more efficient algorithm is implemented) may not be visible from class, sequence or statechart diagrams, suggesting that model-based and code-based approaches are complementary. These issues will be discussed in further details in the following sections.

In this paper, we focus on automating regression test selection based on architecture and design information represented with the Unified Modeling Language (UML) and traceability information linking the design to test cases. Our focus on the UML notation is a practical choice as it has become the industry de-facto standard. The original test set from which to select can contain both functional and non-functional system test cases. From a UML standpoint, functional system test cases test complete use case scenarios.

The rest of the paper is structured as follows. Since UML is only a notation, we first precisely describe the assumptions we make regarding the way it is used (Section 2). The following section describes the detected changes from UML class and use case/sequence diagrams as well as their impact on the classification of test cases (Section 3). To do so, we provide both intuitive definitions and a formal mapping using set theory. In Section 4, we analyze our model-based regression test selection strategy in the light of the framework proposed in [27], though this framework has been originally defined for white-box regression test selection strategies. Section 5 briefly introduces the functionality of the Regression Test Selection Tool (RTSTool) we built based on the principles introduced in Section 3. Sections 6 and 7 report the details of case studies and further discuss related works, respectively. Conclusions and future directions are then drawn in Section 8.

2. Assumptions on the use of the UML notation

This section focuses on the testability of UML diagrams, that is the extent to which they can be used to support test automation. As UML is only a notation, we need to make a number of assumptions about the way UML diagrams are used [6] to automate their analysis and facilitate traceability between test cases and the UML models. Though what we write in this section should not be surprising to the experienced UML practitioner, it needs to be clarified so as to automate our regression test selection methodology.

2.1. Consistency assumption and design by contract

First of all, we assume the different UML diagrams we rely on, i.e., use case, sequence and class diagrams, are consistent with each other. Otherwise, one cannot guarantee the validity of any UML-based analysis. For instance, if an operation has been deleted from a class in the class diagram, we assume the sequence diagrams in which the operation appears in the label of a message have been updated. Consistency checking can be easily implemented [4] and is a separate issue from the focus of the current paper. Note that modern modeling environments, such as Rational Software Architect [14], already support such consistency analysis.

Following the well-known Fusion method [9] and the Design By Contracts principles [20,21], we assume that class operations are described by providing their precondition and postcondition. In the context of UML, such contracts are typically described using the Object Constraint Language (OCL [32]). We also assume that class invariants are provided in OCL.

2.2. From use cases to sequence diagrams

Another issue is related to the combined use of use cases and sequence diagrams. We assume that with each use case we associate a unique sequence diagram specifying the possible object interactions that realize all possible use case scenarios. In practice, scenarios can be specified across several sequence diagrams to improve their readability but we assume there is only one, complete sequence diagram.[1] We also assume, following best practices, that sequence diagrams be named [16], and that they be named after the use cases they realize. As a notational convention, sequence diagram A refers to the sequence diagram for use case A.

Use cases relate to each other in the use case diagram by means of include, extend and generalization relationships [2]. For instance, in an Automated Teller Machine (ATM) system, the `DoTransaction` use case includes use case `InsertCard`. In other words, common functionalities across use cases are factorized out to reduce complexity in the use case diagram and use case textual descriptions. This also results in simpler sequence diagrams that focus only on the event flows of the corresponding use cases rather than on the event flows of included or extension use cases. Definitions for include and extend use case relationships (with extension points [24]) allow a clear identification of when in the corresponding flow of events a use case invokes another use case.[2] Moreover, UML 2.0 provides a mechanism, namely Interaction Uses, to translate use case relationships into sequence diagrams [24]. This mechanism applies to both include and extends use case relationships and allows: (1) Complete scenarios possibly exercising several use cases

[1] There is no technical or theoretical difficulty in merging sequence diagrams modeling different scenarios of a same use case into one complete sequence diagram.

[2] An include relationship between use cases means that the base use case explicitly invokes another use case at a location specified in the base. An extend relationship between use cases means that the base use case implicitly invokes another use case at a location specified indirectly (i.e., conditions, extension points) by the extending use case [2].