Automatic Test Suite Evolution

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Mehdi Mirzaaghaei

under the supervision of
Mauro Pezzè

November 2012

Dissertation Committee

Walter Binder	University of Lugano, Switzerland
Cesare Pautasso	University of Lugano, Switzerland
Darko Marinov	University of Illinois at Urbana-Champaign, USA
Paolo Tonella	Fondazione Bruno Kessler, Trento, Italy

Dissertation accepted on 9 November 2012

Research Advisor	PhD Program Director
Mauro Pezzè	**Antonio Carzaniga**

I certify that except where due acknowledgement has been given, the work presented in this dissertation is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the dissertation is the result of work which has been carried out since the official commencement date of the approved research program.

Mehdi Mirzaaghaei
Lugano, 9 November 2012

To Shima

Abstract

Software testing is one of the most common approaches to verify software systems. Despite of many automated techniques proposed in the literature, test cases are often generated manually. When a software system evolves during development and maintenance to accommodate requirement changes, bug fixes, or functionality extensions, test cases may become obsolete, and software developers need to evolve them to verify the new version of the software system. Due to time pressure and effort required to evolve test cases, software developers do not update test cases often and many test cases become quickly obsolete. As a consequence, the effectiveness of the test suites drops over time.

In this thesis, we propose a new technique for automating test suite evolution aiming to reduce both the developers effort and the costs of software evolution. To understand the evolution of test cases, we conducted an empirical study on real-world software systems to gain an insight on how software developers evolve the test cases. Our study shows that many test cases of software systems are similar. These similarities lead developers to follow common activities for repairing and generating test cases by using some common processes. Based on this observation, we identify some of *Test Reuse Patterns* that can be applied to fix many test cases and generate new test cases.

In this thesis we define five *Test Reuse Patterns*: change of method declaration, extension of class hierarchy, implementation of interface, introduction of overloaded methods, and introduction of overridden methods. We propose a framework for generating and repairing test cases for evolving software by automating *Test Reuse Patterns*. The framework, called Test Care Assistant (TCA), repairs test cases that do not compile after the code changes and generates test cases when new classes or methods are added to the software. We evaluated TCA on several open-source projects, and the results suggest that TCA can be successfully adopted in software testing process to help software developers evolve the test cases with less effort and time. The empirical evaluation of the framework shows that TCA can be successfully applied in many cases and that the effectiveness of the test cases generated and repaired by TCA is comparable with

the effectiveness of the test cases written manually by software developers and generated automatically by existing techniques.

This dissertation makes several contributes to the state of the art. First, it introduces *Test Reuse Patterns*, Second, it proposes a framework, TCA, that exploits the *Test Reuse Patterns* to repair and evolve test cases in the software, Third, it performs an empirical study of software evolution conducted on different test suite repositories, Fourth, it implements the TCA prototype, and Finally it performs an empirical evaluation of the applicability and effectiveness of the approach on open-source projects.

Acknowledgements

First, I would like to thank my advisor, Mauro Pezzè. Your expertise and suggestions contributed significantly to this work. I am grateful for giving me the freedom to explore the things that I considered worthy to investigate and your encouragement for my academic growth. I learned a lot from your mature way of thinking. Your support and advice were priceless.

I wish to thank Fabrizio Pastore which advised me much beyond his time as a post-doctoral fellow at University of Lugano. The collaboration with him taught me more than he might think. The many discussions we had influenced significantly the research presented in this thesis. Grazie, Fabrizio!

Special thanks to Walter Binder, Cesare Pautasso, Darko Marinov, and Paolo Tonella for accepting to serve as scientific committee members of my work as well as your detailed feedback on the dissertation proposal. I really appreciate your time and thanks for taking me seriously.

Many thanks to Mehdi Jazayeri for being available for any discussion during my PhD studies and for showing interest in my research in all the stages of my work. I would like to thank other faculty members of University of Lugano specially Carlo Ghezzi, Marc Langheinrich, Michal Young, Fernando Pedone, and Antonio Carzaniga for investing a part of your precious time to teach me new things.

Extensive discussions with Darko Marinov and Danny Dig greatly improved the empirical studies of this thesis. During my visit of Darko's group, I enjoyed discussing my PhD thesis with people in UIUC: Thanks Darko, Danny, Vilas, Milos, Qingzhou, Adrian, Mohsen, Samira, Cosmin, Steve, and Brett.

Thanks to all the graduate students in University of Lugano that I had fun and shared my endeavor with: Amir, Alessandra, Alessio, Alex, Andrea, Cyrus, Giacomo, Giovanni, Jochen, Konstantin, Marcello, Marco, Mattia, Mauro, Mircea, Mostafa, Navid, Nemanja, Nicolas, Nicolò, Paolo, Parisa, Parvaz, Roman, and Saeed: I am thankful for your company.

I am always grateful for support of my family specially my parents. Thanks for your unconditional love, trust, and support. I am proud to be your son. I am

very luck to have the support of two lovely sisters and my brothers-in-law which gave me hope throughout my PhD. I am also thankful for all the encouragement of my parents-in-law.

The last but not the least, a big thank you goes to Shima, for sharing with me an unforgettable time in Lugano. Thank you for your love, encouragement, and patience. Without you I could not have concluded this thesis.

Mehdi Mirzaaghaei
November 9, 2012

Contents

Figures

Tables

Listings

Chapter 1

Introduction

Software testing is the most common practice for validating and verifying software systems. During development, software developers write test cases to check the compliance of the developed applications with their specifications. Due to the complexity of software systems, software testing is an expensive activity, and the cost and complexity of software testing increases when software evolves to reflect changes in the system [PY07].

Testing activities include test case design, execution, and maintenance, which are often performed manually by software developers during the evolution of the system, when software components are added, removed, or changed. After modifying the system, developers re-execute existing test cases to identify regressions in the functionality, repair the existing test cases if they have been broken by the changes performed, and develop and execute new test cases to verify the new functionality. Evolution of test cases is particularly common because of the wide adoption of agile software development processes such as Scrum [SB02], which are characterized by frequent changes of requirements and functionality and at the same time by the early definition of test cases, which consequently should be adapted quite often during the lifespan of a software system. In this dissertation, we use the term *test evolution* to refer to the activities performed to keep the test cases up to date while software evolves, including test repair and generation of new test cases.

The process of evolving test cases is often performed manually by software developers, with the consequence of an increase in the software costs. Manual adaptation of test cases is also time consuming: software developers rewrite the test cases according to changes in the API or in the software specifications, or they add new test cases to cover new functionality.

Researchers developed several techniques to reduce the efforts of test case

generation and evolution. Many techniques generate test cases by using approaches like symbolic [Kin76], concolic [GKS05, SMA05], or random execution [PE07]. These techniques are effective, but come with some limitations: (1) they automatically generate test cases that are difficult to understand for software developers, (2) they do not leverage domain information, (3) they cannot identify the necessary setup actions to run the test cases, (4) they generate many invalid test inputs that are not desirable for the developers, and (5) they do not provide oracles with the generated test cases. Therefore, the software developers need to inspect and understand the generated test cases to complete and execute them [JLDM09]. Model-based testing generates both test inputs and oracles from software specifications, and thus does not require developers to modify or complete the generated test cases [DNSVT07]. Unfortunately software engineers usually define specifications only for few components, often the critical ones, thus the applicability of model-based techniques remains limited in practice. In practice, although there are several test generation techniques available, but software developers still largely rely on manual definition and implementation of the test cases [XKK+10].

The goal of this dissertation is to define a framework to automate test suite evolution. Our framework uses exiting test cases to evolve test suites by repairing obsolete test cases and generating new test cases while software evolves. We argue that existing test cases encapsulate domain knowledge and thus can be reused to generate test setup actions, inputs, and oracles without requiring complete specifications. We reuse existing test cases and automatically adapt both test inputs and test oracles to reduce developers effort thus alleviating the limitations of current test case generation techniques.

We manually inspected different versions of several open-source systems and found out that software developers often reuse and adapt existing test cases rather than writing the test cases from scratch, to evolve test suites. We identified frequent *actions for adapting test cases* that developers commonly apply to correct and generate test cases, hereafter *Test Reuse Patterns*, and we relied on the identified actions to define *algorithms for evolving test cases* as a solution to support software developers in the evolution of test cases.

The *Test Reuse Patterns* are the basis for the definition of algorithms that generate test cases by automating the activities captured by the reuse patterns. For example, to generate test cases for classes that extend a class hierarchy, software developers usually apply the following test adaptation pattern: copy the test cases of another class in the hierarchy, replace the instances of the original class with the instances of the new one, replace old test inputs with new valid test inputs for new class, and update test oracles. A concrete example of this *Test*

Reuse Pattern which is called "*extension of class hierarchy*" is shown in Listings 3.5 and 3.6. Our approach automates these activities to generate test cases similar to the test cases that the developer manually writes.

The availability of test cases for the software under test is a prerequisite for the automatic generation of new test cases with our proposed approach. We expect that the developers generate a set of test cases for the original version of the system either manually or with automated test generation tools. Then, software developers can apply the framework proposed in this dissertation to generate test cases for added or modified functionality during the evolution of the system. While agile software development practices such as test-driven development urge the developers to write test cases before the code, the test cases are not maintained properly in practical development environments.

We evaluated our framework on a set of open-source case studies written in Java. We compared the test cases produced by our technique with the test cases written by the developers of those projects and with the test cases automatically generated by some of best known state-of-the-art test generation tools such as Randoop [PLEB07], CodePro [Cod12], and EvoSuite [FA11a].

The results of our experiments suggest that our framework can generate and repair many test cases comparable with test cases written by the developers and generated by the state-of-the-art test case generation techniques. Moreover, the test cases generated by our framework are as readable as the ones written by software developers, thus they result tend to be more understandable and concise than the ones automatically generated by test case generators.

1.1 Research Hypothesis and Contributions

The main hypothesis that motivates our work is:

> *Existing test suites contain domain knowledge that can be automatically reused to generate new test cases and repair existing test cases*

The first part of the hypothesis describes the main intuition of this dissertation, which is that existing test cases are a good source of information to generate test cases for exercising evolving software, but reusing existing test cases is neither well exploited in the current test case maintenance techniques nor fully investigated in the literature. While best software development practices do not recommend using anti-patterns such as code clones, in paratactical development environments, the developers clone the code to extend the functionality of software systems [JS09, GJS08]. We can reuse the knowledge of developers

encoded in the existing test cases to test modified and extended part of software systems. This intuition comes from the observation of similarities of the new test cases generated for the modified software and the changes in the original test cases. When the developers write new test cases for extended functionality or repair existing test cases, they reuse the test cases available for the former version of the software.

This dissertation contributes to the state-of-the-art by:

- Identifying a set of *reuse patterns* for test cases, i.e., activities commonly performed by software developers to generate test cases by reusing and adapting existing ones [Mir11, MPP10, MPP12].
- Defining a *framework* to automatically generate test cases for evolving software systems based on the identified *Test Reuse Patterns*. The underlying idea is to reuse existing test cases as the source of information to generate test cases for functionality modified and added to the system during its evolution [MPP11].
- Providing experimental evidence of the effectiveness of the approach in repairing over 138 test cases and generating test cases for over 700 classes and 2400 methods in 5 open-source projects. The evaluation shows that TCA can be effectively used in software maintenance processes and reduce the efforts of the developers by allowing to evolve test cases automatically [MP11, MPP12].

1.2 Scope of Research

In this dissertation we consider white-box test suite generation and evolution which implies availability of the internal structure of the software, i.e., its source code. We focus on fine grained changes in software systems and consider changes in the structure of source code. Our technique applies during the development and maintenance phases of the software development lifecycle and we assume that software developers generate test cases while developing software (we call the test cases generated in this stage as *developer test cases*).

We use programs written in Java as proof of concept; however, our approach is applicable in general to any typed object-oriented language. We consider unit test cases written for a class under test, and we use JUnit[1] as test automation framework.

[1] http://www.junit.org

The approach presented in Chapter 4 relies on static def-use analysis of the classes under test, and operates at byte code level. The approach generates test cases in Junit format by modifying the source code of the test case that should be available to the tool. The approach presented in Chapter 5 uses both the AST of the classes under test and test cases to generate test cases from existing test cases, thus requires the AST of the classes and the source code of the test cases, but not the source code of the dependent libraries of the systems under test.

1.3 Structure of the Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 overviews several techniques related to test suite evolution. We define a taxonomy that we use to classify techniques that generate and maintain test suites. We discuss the state-of-the-art techniques on automatic test case generation, augmentation, and repair.

- Chapter 3 introduces the test repair and adaptation framework that we call *Test Care Assistant*, TCA, which automates test suite evolution by using a set of test reuse patterns. We define a catalogue of test reuse patterns for test suite repair and adaptation, and illustrate each pattern with examples taken from open-source projects.

- Chapter 4 explains the details of the algorithms developed for the TCA framework to repair test cases. TCA repairs compilation errors induced by parameter type change, parameter addition, parameter deletion, and return type change.

- Chapter 5 introduces the details of the algorithms that we developed for generating test cases by adapting existing test cases, following different *Test Reuse Patterns*. The chapter discusses the algorithms for four *Test Reuse Patterns* that apply when adding a class to a hierarchy, implementing new interfaces, overriding methods, and overloading methods. TCA follows a similar process to generate test cases for all the identified *Test Reuse Patterns*.

- Chapter 6 presents a prototype implementation of TCA that we developed as an Eclipse plugin. Our prototype tool supports the developers in repairing and generating test cases for evolving software. We explain the

underlying implementation details and the technology that we used in the prototype.

- Chapter 7 presents the empirical results of the experiments that we conducted on several case studies by applying our prototype on open-source projects. This chapter compares the TCA framework with the competing state-of-the-art techniques.
- Chapter 8 concludes this dissertation by discussing our approach, summarizing our contributions, and outlining future research directions of this research work.

Chapter 2

State of The Art

During the lifetime of a software system, software developers maintain test suites by generating test cases for new functionality and by repairing the test cases that become obsolete due to changes in the specification of the existing components.

Many researchers focus on the reduction of test maintenance costs by defining techniques to automatically generate and repair test cases or prevent problems caused by changes in existing components, which typically occur after system refactoring. This Chapter overviews the state-of-the-art and highlights the limitations of current approaches. In particular, this chapter first overviews the current approaches to automatically generate test cases, then introduces the techniques that reduce cost of test maintenance such as test suit repair and regression testing, and finally presents approaches that propose new test processes and test maintenance techniques.

2.1 Test Case Generation

We classify the techniques that automatically generate test cases in five main categories: random testing, symbolic execution, search-based, model-based, and heuristic-based techniques.

2.1.1 Random Test Case Generation

Random test case generation produces test inputs by randomly sampling the input space. Random testing generates test cases very efficiently, is simple to implement, and scales in large software systems [CLOM08, GKS05, CKMT10]. Pure random test generation tends to produce many illegal test cases and can

hardly generate test cases that exercise particular behaviors. To improve the effectiveness of random testing, researchers have extended the approach in several ways.

Pacheco et al. [PE07, PLEB07, REP+11] propose directed random testing that explores the input space by selecting random inputs that do not cause the software to raise exceptions. Directed random teasing uses a feedback directed approach to generate a sequence of method calls that creates and mutates objects, plus an assertion about the result of a final method call. Directed random testing builds a test by randomly selecting a method or a constructor, using previously computed values as inputs. Directed random testing only builds legal sequences of method calls, and uses the feedback obtained from executing the sequence as it is being constructed, to guide the search toward sequences that yield new and legal object states. The implemented technique is called Randoop and works with applications written in Java [PE07] and .Net [PLB08], and is extensively evaluated on large applications. Randoop has been recently extended to generate maintainable test cases [REP+11].

Andrews et al. [AML10, ALM07, AM09] use genetic algorithms to improve the effectiveness of random testing. They generate a pure randomized test suite for the software under test and place them in a pool of objects. They design a genetic algorithm to find the best objects from the pool with the goal of optimizing test coverage. The results of their evaluation suggests that the technique reduces the size of randomized test suite while achieving the same coverage. Designing the genetic algorithm and tuning it requires some prior knowledge and the approach is not yet applied to large scale software systems.

Chen et al.[CKMT10, CM07, CM05] propose another way of trimming random inputs by systematically guiding random generated candidates to cover contiguous areas that are empirically identified as likely to be faulty. The approach uses adaptive mechanisms to improve the failure-detection effectiveness of random testing. Adaptive approaches systematically guide or filter randomly generated candidates to take advantage of the likely presence of failure patterns. Empirical studies identify faulty areas of software systems on numerical input domain and guide the input generator to exercise those software sections. For instance, the algorithm guides random test generator to generate inputs that lead to division by zero. The effectiveness of the approach is shown on small numerical programs. Despite of extensive discussions, the scalability of the approach remains challenging.

Ciupa et al. [CLOM08] use another approach to prune generated random inputs. The approach generates candidate inputs randomly and at every step selects the test inputs that are more distant from the already generated inputs.

The distance is measured as the elementary distance between the direct values of the objects, a distance between the types of the objects, and recursive distances between the fields of the objects. This way they make sure the generated test cases evenly test the whole program.

Although random approaches are argued as effective as systematic approaches, random approaches still face challenges in practice to generate sequences for achieving target states [VPP06a, DN84]. The reason is that the probability of randomly generating test cases that reach some specific target states is very low. When the source code and the specifications are unavailable or incomplete, random testing may be the only practical choice. Random testing has been used extensively both as a testing method itself, and as a core part of other testing methods. Random test generation techniques do not deal with evolution in software systems. However a recent work tried to generate maintainable test inputs that can be used when the software is evolving in an industrial settings [REP+11].

2.1.2 Symbolic and Concolic Execution

Symbolic execution [Kin76] is a technique that uses the program code to derive a general representation of its behavior. The program is executed on symbolic rather than concrete inputs, and a set of constraints on the symbolic inputs is collected along an execution trace. A constraint solver is then used to generate test inputs that satisfy the symbolic constraints. The resulting test inputs are guaranteed to force the program execution along the path chosen by the symbolic execution. There are many techniques that use symbolic execution to generate test cases for methods and classes in object-oriented software. For example, researchers [Kin76, GKS05, PMB+08, SRRE08, TBV07, VPK04] identify the inputs that cover the feasible elements of a program by solving the path conditions of the program paths (typically using an automatic theorem prover). Unfortunately for large and complex systems, it is computationally intractable to precisely maintain and solve the constraints required for generating test cases with symbolic execution.

To overcome some limitations of symbolic execution, researchers proposed dynamic symbolic execution (a.k.a concolic execution) that combines symbolic with concrete execution [God07, GKS05, MS07, SMA05, TDH08]. The approach differs from the traditional symbolic execution because it executes a program on concrete inputs while simultaneously recording symbolic path constraints. Which branch to take is then determined by concrete values. By adjusting the

path constraints, usually by negating one of its branch conditions, standard constraint solving techniques [dMB08] can produce new concrete values that force the program execution down a different program path. Pex is one of the most popular symbolic execution engines developed at Microsoft Research for analyzing .NET code. It uses the Z3 constraint solver [dMB08], which provides decision procedures for most constraints encountered in the .NET intermediate language as well as constraints involving low-level pointers and memory references. CUTE [SMA05] and DART [GKS05] are some other techniques that use concolic execution to reduce some of the constraint solving limitations by combining path constraints evaluation with random generation of test inputs [GKS05, SMA05, CLOM08, AML10].

Bounded exhaustive testing reduces some of the limitations of test input generation techniques by requiring specifications of the valid test inputs from developers and by exhaustively generating inputs that satisfy these conditions in a subset of input domain [CYK+05, SYC+04, GGJ+10, BKM02]. However, target sequences involving classes from real-world applications often require longer sequences beyond the small bound handled by bounded-exhaustive approaches.

2.1.3 Search-Based Test Generation

Search-Based Software Testing (SBST) has recently attracted a lot of attentions in research community. SBST uses meta-heuristic algorithms[1] to automate the generation of test inputs that meet a test adequacy criterion. Branch coverage is one of the most widely-studied test adequacy criteria in SBST [McM05]. Search based techniques that are used in the area of test data generation utilize hill climbing, simulated annealing, and evolutionary algorithms (like genetic algorithms). Hill climbing is a local search algorithm that tries to improve one solution by exploring the neighborhood of a first solution. Hill climbing is fast, but can achieve sub optimal results, also called local optima. Simulate annealing improves on hill climbing by exploring new and potentially better solutions by randomly choosing new points thus avoiding local optimum. However it might take an infinite amount of time to reach global optimum because the algorithm recursively backtracks if global optimum is not reached. Evolutionary approaches [HM10, BM10, IX08, LMH10, Ton04] maintain a population of solutions and produce better generation of solutions by combining the maintained solutions. Specifically, they accept an initial set of sequences and evolve those

[1] meta-heaurisitic algorithms optimize a problem by iteratively improving a candidate solution with regard to a quality measure

sequences to produce new sequences that can generate desired object states. Harman and McMinn further investigated some global search techniques used in evolutionary approaches [McM04, HM10]. Here we briefly discuss some of these test generation techniques.

Tonella proposes a test case generation technique based on genetic algorithm to automatically produce test cases for the unit testing of classes [Ton04]. Test cases include information on which objects to create, which methods to invoke, and which values to use as inputs. The proposed algorithm mutates the inputs aiming to maximize a given coverage measure.

Fraser and Arcuri propose a search based technique, and accompanying tool called EvoSuite, that automatically generates test cases with assertions for classes written in Java code [FA11b, FA11a]. They combined static data flow analysis and genetic algorithm that generates and optimizes whole test suites towards satisfying a coverage criterion. Optimizing with respect to a whole test suite coverage criterion rather than individual coverage goals alleviate the problem that derive from the infeasibility of individual coverage goals. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarizes the correct behavior; these assertions allow developers detect deviations from expected behavior and capture the correct behavior to protect against future defects breaking this behavior, i.e., regressions. Evosuite obtains reduced sets of assertions by generating a set of oracles and reducing them by their mutation killing scores. The assertions of the test case can only be used for regression testing since it extracted from code but not from specification.

2.1.4 Model-Based Testing

Model-Based Testing (MBT) relies on explicit behavioral models that encode the intended behaviors of a software under test and the behavior of its environment. Test cases are generated from one of these models or their combinations, then executed on the software unit or system under test. The use of explicit models is motivated by the observation that traditionally, the process of deriving test cases tends to be unstructured, not reproducible, not documented, lacking detailed test design, and dependent on the ingenuity of single engineers. The idea is that artifacts that explicitly encode the intended specification of software under test and possibly environment behaviors can help mitigate these problems.

There is a lot of research work done in the area of model based testing that use different artifacts of software development process such as UML diagrams, finite state machines, Z specifications [Jac90]. Model based testing approaches

evaluate the test generation effectiveness by a variety of criteria such as structural model coverage, data coverage, requirement-based coverage, fault based coverage. Despite the availability of many model based techniques, only few techniques find industrial applications and are extensively evaluated. The models used for generating test cases are often not integrated within the software development process [Utt05] and extra work is required to extract the models and keep them updated while software evolves. Moreover, model based techniques require knowledge about the modeling language, the coverage criteria, the generated output format, and the supporting tools which make the use of MBT difficult in real world scenarios [DNSVT07, CPU07, Muc07, FAW07].

2.1.5 Heuristic-Based Approaches

Recently, researchers propose to reuse application code to generate new test cases toward improving test coverage [TXT+09, dPX+06, VPP06b]. The goal of these techniques is to reduce the number of invalid test cases generated with automatic techniques without affecting their effectiveness. These techniques identify and remove invalid inputs by extracting method call sequences that are not present in the source code of other software systems that use the software module under test.

MSeqGen [TXT+09] and Seeker [TXT+11] mine client code bases and extract frequent patterns as implicit programming rules that are used to assist in generating test cases. Such approaches can be sensitive to the specific client code, thus the results depend on the quality of client code base provided by the code search engine.

DyGen [TdHTW10] generates test cases by mining dynamic traces recorded during program execution. DyGen uses symbolic execution to find test cases that cover new paths that are not already covered by the existing test cases. DyGen achieves better coverage than seeding test cases, however, there are no information available on the overheads on running, recording, and symbolically executing programs under tests.

Although heuristic-based approaches can reduce the number of generated test cases while improving code coverage, their applicability remains limited. These techniques still generate test cases that are often difficult to be understood, thus forcing software developers to spend time in understanding the generated test cases.

Automatic test case generation techniques often do not identify the setup actions necessary to execute the test cases, and tend to generate a huge amount of test inputs without distinguishing among valid and invalid inputs thus causing

invalid failures. Since the automatically generated test inputs are too complex to be understood by software developers, practical applicability of these techniques is limited to regression testing or detection of unexpected exception conditions.

2.2 Regression Testing

Regression testing is the activity of testing software after it has been modified to gain confidence that (1) newly added and changed code behaves correctly and (2) unmodified code does not misbehave because of the modifications [HO08]. The problem of deriving regression test cases includes some subproblems: test selection, test prioritization, test augmentation, and test minimization.

2.2.1 Regression Test Selection

When a software changes, the developers need to know which test cases to update and which test cases to generate. The process of selecting the test cases to be executed on the new version is called regression test selection. Although executing all test cases is a possibility, it can result in a waste of testing resources because of executing unnecessary test cases. An analysis of the changes between original and modified software can select a subset of test cases to rerun on the modified software. Regression test selection techniques identify the part of software that need to be retested by creating a representation of the changes. Some techniques use source code representations, referring to entities such as statements [VF97], branches [RH97], control and data dependencies [HR90, GHS96, Bin97], functions and non-executable components [CRV94, NMS+11], and program paths [Bal98a] or traces [RGJ06]. Other techniques use representations based on software artifacts other than the source code, such as UML diagrams [BLH09, BLO03], architectural models [MDR06], and program spectra [XN05]. There are similar techniques on different types of languages, such as object-oriented languages [HJL+01, HMF92, OSH04, RRST05] and aspect-oriented languages [XR07], and for different types of programs, such as graphical user interfaces (GUI) [MNX05, MS03, Bin97, Mem04, Mem04] and component-based systems [ZRWS06].

2.2.2 Test Suite Prioritization

After selecting a subset of test cases to be re-executed, the number of test cases might still be infeasible for available resources and timings. In this case, it can

be useful to order the test cases according to some criteria. This activity is called test case prioritization. The criteria can refer to the code coverage or the estimated fault detection ability. These techniques use various kinds of information about original and modified software such as requirements [HS05], source code [ST02, JH03], models of the software [KTH05, CPU02], time to run test cases [WSKR06], and interactions among events [BM07]. Some research exploits techniques such as evolutionary algorithms to determine the test case ordering [LHH07]. Jonatan et al. [JLM11] propose a technique to prioritize regression test suite of multi threaded software.

2.2.3 Test Suite Augmentation

The test cases written for original software may not be enough to adequately exercise modified or new functionality. In these situations, some test cases need to be created and added to the test suite. This activity is called test suite augmentation. Like regression test selection, test suite augmentation techniques also generate some models of original and modified software and assess the differences between them to identify the characteristics of test cases to be added. The characteristics of test cases can be extracted with data-flow [LM90, Tah92], control-flow [RH97, LS92], and both data-flow and control-flow techniques [RH94, Bal98b, Bal98a, GHS96]. More recent techniques use symbolic execution related to the changes as a way to compute and characterize differential behavior in original and modified software [XR09, XKK+10, ASC+06, SCA+08]. Xu et al. generate test cases that cover execution paths of the modified software not covered by existing test cases [XR09, XKK+10]. They first identify the test cases affected by the changes in the software, then run the existing test cases to determine the branches not covered during test execution. They use concolic execution to cover remaining branches by negating the path conditions.

Similarly, Santelices et. al. propose an algorithm that indicates which execution paths should be covered by new test cases of the application to verify the behavior of the modified software [SCA+08, ASC+06]. They use symbolic execution to determine which inputs generate different outputs in two consecutive versions of the software, then suggest the test requirements to cover the modified execution paths.

2.2.4 Test Suite Minimization

Adding test cases to the test suite improves the effectiveness and supports the evolution of test suites along software evolution, but the size of the test suites

grows incrementally, resulting in insufficient resources to run and maintain the test cases. To address this issue, some techniques identify and remove test cases that are redundant, for example, if they exercise the same behaviors in the software. This activity is called test suite minimization. Researchers have extensively investigated the problem of test-suite minimization by identifying test cases that are redundant according to some criteria and removing them. Researchers have proposed techniques that operate directly on programs source code [HGS93, JH03] and use coverage information to identify redundancy. Several researchers have investigated limitation of test suite minimization with different results [MM07, RHOH98, WHLB97]. They argue that although minimizing test suite significantly reduce the size of a test suite, but also reduces fault-detection ability of the test suite.

Regression testing approaches work only in the case of modifications of existing methods, but do not handle the changes that derive from introduction of new components to the system. Among subproblems of regression testing described above, this dissertation is closer to test augmentation techniques with two essential differences: (1) test augmentation techniques cover alternative paths introduced in the modified software, while this dissertation focuses on the addition of new functionality to the software systems, (2) test augmentation techniques use test case generation techniques to cover alternative paths while this dissertation focuses on generating test cases by reusing existing test cases and adapting them for new functionality.

2.3 Test Suite Repair

When software evolves, some of the test cases that are designed to test the original software become obsolete. Obsolete test cases are test cases that either cannot be executed, or execute, but do not check the functionality of the modified software. After each modification, the developers need to examine the obsolete test cases to make them compatible with the modified software or fix the bug. This process is called test suite repair. Currently most test cases are repaired manually by the developers. Since the developers are not rewarded to update test cases, they often simply remove obsolete test cases from the test suites, thus reducing the effectiveness of test suite [REP+11]. Therefore, during software evolution, the test suites become less and less effective. This shows the importance of automatically repairing test cases.

Some researchers focused on repairing obsolete test cases for GUI applications, while others focused on repairing the oracles of test cases. We are not aware of any technique that deal with interface changes in general unit test cases.

2.3.1 Repairing GUI Test Cases

In the context of GUI testing, software developers create test scripts using capture-and-replay tools [HZ93]. Such scripts may become invalid due to even simple modification of the interfaces of the system, such as graphical widgets and programming APIs.

Several techniques have been proposed to correct GUI test cases broken by changes in the system. Memon et al. propose a technique that repairs obsolete GUI test cases by representing changes with Event Flow Graphs (EFGs) [Mem08, MNX05, MS03]. An EFG models all possible event sequences that may be executed on a GUI. An EFG contains nodes (that represent events) and edges. An edge from node x to y means that the event represented by y may be performed immediately after the event represented by node x. Memon et al. developed an automated tool, called GUI ripper, that traverses the GUI by opening the windows and extracting all the widgets, properties, and values. GUI ripper generates partial test sets. Test designers should review the generated EFGs, and produce the missing parts of EFGs by means of classic capture and replay tools.

Fu et al. propose a technique to repair GUI-based test cases for specific scripting languages like JavaScript [XGF08, FGX09]. The technique models GUI of both original and modified software by navigating the GUI screens under test and uses an automated tool, called GUI Modeler, to obtain information about the structure of the GUI and the properties of individual objects. The GUI Modeler outputs GUI trees for the original and the modified software. The comparator with the help of test engineer determines the modified objects in GUI trees. The Script Analyzer analyzes test scripts, utilizing modified objects, to determine possible failures in test scripts. Test engineers review the potential failures and modify the original test script for the GUIs of the original version so that it can test modified GUI of the software. This technique relies on the intervention of the users who ultimately recommend the possible repair to the user based on the information provided by the generated models. The required effort highly depends on the expertise level of the user and the size of the software under test.

All the above approaches, repair the test cases after they become obsolete. On the contrary, Daniel et al. propose a technique to record changes in the GUI

of applications and later apply them to test cases. In fact this technique tend to record the process of modifying the GUI by the developers. This process is called the refactoring of application GUI [DLM+11]. The approach is preliminary, but can significantly improve the GUI test suite repair if suitably implemented.

The approaches presented in this section are specifically designed for GUI test cases, and can not be easily generalized to other classes of software systems. This dissertation focuses on repairing general purpose test cases.

2.3.2 Repairing Test Oracles

When changes cause the failure of the assertions in existing test cases, developers should inspect the failures. The failures are either caused by regressions, or by changes in the intended system behavior. In the former, the developers must revise the application code to make the test cases pass. In the latter, the developers must repair the broken test cases or remove them from the test suite. Repairing broken test cases is time consuming, therefore, automating oracle repair can help developers keep the test cases up to date at a lower cost.

Daniel et al. propose a technique, called ReAssert, that suggests repairs for failing unit test cases while retaining their power to detect regressions [DJDM09]. When test cases fail their assertions, ReAssert suggests changes to the test case that cause the tests to pass. If the suggested repairs match the developers' intentions, ReAssert repairs the test case with little overhead. The ReAssert repair process starts when the user chooses a set of failing test cases to repair by following multiple strategies. For each test, ReAssert iteratively attempts to repair the code until the test case passes, no strategies apply, or the iteration limit is reached. To repair a single failure in a test case, ReAssert first instruments the test classes to record values of method arguments for failing assertions. It then re-executes the test and catches the failure exception that contains both the stack trace and the recorded values [DGM10]. Next, ReAssert traverses the stack trace to find the code to repair by examining the structure of the code and the recorded values. It finally recompiles the code changes and repeats these steps if the test has another failure.

A recent empirical study shows that in practice the techniques that focus on repairing oracles such as ReAssert can only target a small subset of broken test cases [PSO12]. Moreover, the suggestions proposed by ReAssert must be validated by software developers. Since ReAssert modifies test cases to make them pass, it could erroneously mask a failure. Another practical limitation

is that ReAssert only repairs test oracles while it cannot fix test inputs, thus developers still need to manually update test inputs.

2.3.3 Refactoring Techniques

Refactoring techniques are designed to restructure a software system without altering the external behavior, while improving internal structure of the code [Opd92]. In the context of test case maintenance, the behavior of the test case may be altered when software evolves.

Since current integrated development environments support test case creation and the test cases are maintained in the same project as the source code, refactoring tools can access the test cases as well as source code for any changes in the interface of the system. Therefore, automatic refactoring techniques can be partially used to repair test cases although they are not specifically designed for test maintenance [MT04]. Refactoring techniques can prevent simple errors by automating some of the possible refactoring activities like moving or renaming methods, modifying class hierarchies [SS04], or improving concurrency [DME09] and reentrancy [WST09]. Unfortunately, common refactoring practices like adding new parameters to methods [XS06] are only partially automated by existing tools and techniques. Some tools, for example, ReBa [DNMJ08] and Eclipse[2], avoid compilation errors caused by parameter changes only when the modified parameters can be replaced by default values, thus software developers still need to adjust test cases manually. This dissertation provides a framework to improve the test repair from simple program refactoring to full test suite repair.

In this Chapter we provide an overview of the state-of-the-art techniques related to this dissertation in three main categories of test generation, test repair, and regression testing. We discuss the advantages and shortcomings of these techniques as well as the differences between current techniques and our proposed technique.

[2]www.eclipse.org

Chapter 3

Test Evolution

Software developers need to update test suites whenever they change or extend software functionality. To save maintenance effort, software developers reuse and adapt existing test cases instead of rewriting them from scratch. For example, to write test cases of a class that extends a class hierarchy, software developers often reuse and adapt the test cases of another class in the same hierarchy. Software developers behave similarly when they need to repair test cases broken by API changes. When a change in a method declaration like the change of a parameter type causes compilation errors in the test cases, software developers identify the values to initialize the modified parameter by inspecting the original and the modified methods.

By investigating different versions of open-source software systems, we identified several *Test Reuse Patterns* that the developers apply to repair and reuse existing test cases. In our analysis we manually inspected the test cases written by software developers during the evolution of several software systems. We put particular attention in identifying similarities among test cases for classes that implement the same interfaces or extend the same classes as well as test cases for modified methods. The inspection led us to identify several *Test Reuse Patterns* applied by software developers when they repair test cases after changes in method declarations, extend a class hierarchy, implement an interface, overload a method, and override a method.

This chapter provides an overview of our novel approach to evolve the test suite automatically during software evolution. We present the overall building blocks of our framework and describe the usage scenario, then we provide a catalog of *Test Reuse Patterns*.

3.1 TestCareAssistant

To support the evolution of software systems, we propose a framework, *Test Care Assistant* (TCA), that automates the generation and maintenance of test cases by applying algorithms that implement common *Test Reuse Patterns*. Figure 3.1 shows the overall structure of TCA framework. TCA receives as input the original and the modified software plus the test cases for the original software version. As a result TCA generates the test cases for the modified software. To this end, TCA implements the algorithms designed for automation of *Test Reuse Patterns*. The algorithms for repairing and adapting test cases share the same structure with slight customization for each specific *Test Reuse Pattern*.

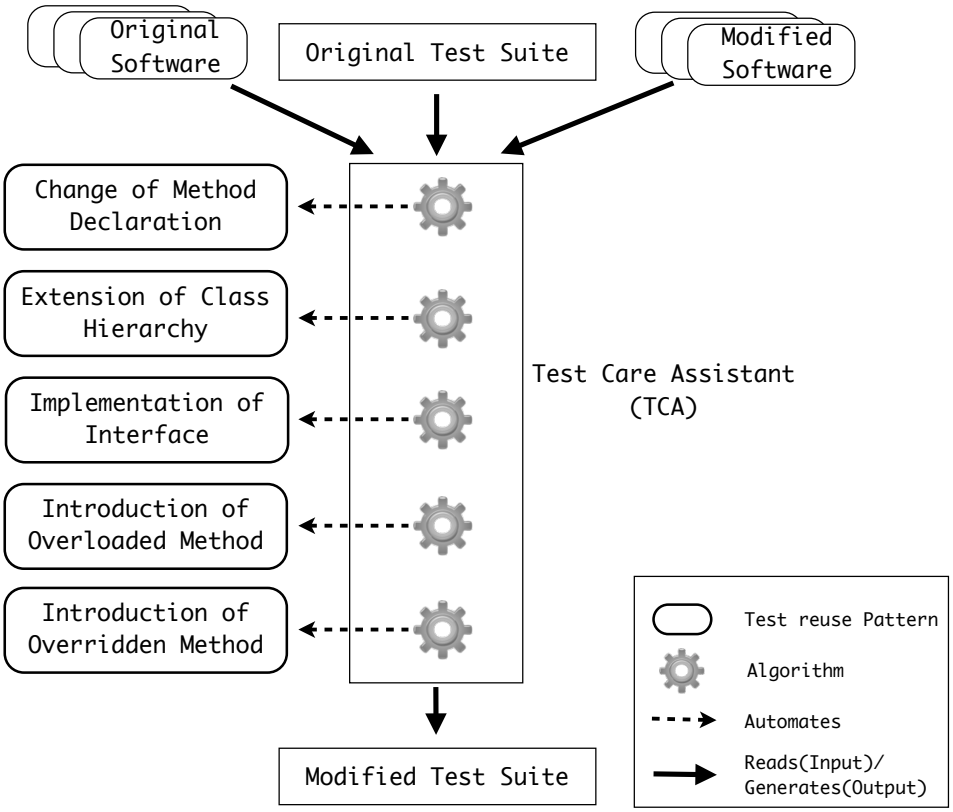

Figure 3.1. Test Care Assistant (TCA)

TCA generates test cases by reusing and adapting existing test cases written for other classes. This characteristic makes the test cases easier to understand, thus software developers can easily modify the generated test cases if necessary. In fact, the test cases adapted by TCA can serve as a starting point for the developers to write the test cases for modified software.

TCA can either repair existing test cases or generate new test cases. It repairs test cases that are affected by changes in the method declarations. In particular it can handle changes in the return type, changes in the parameter type, parameter addition and removal of parameters. TCA does not consider the types of changes in method declarations that are already addressed by existing techniques, like method renaming or parameter swapping. It generates test cases for extension of class hierarchy, implementation of interface, introduction of overloaded method, and introduction of overridden method. In all these cases, the new code shares some elements with the source code of the existing software elements. We expect that TCA will be applicable to other type of changes where there are shared interfaces between the extended and the existing functionality. TCA cannot work when the new functionality does not share any interface element with existing modules.

3.2 Test Reuse Patterns

Software developers often reuse existing test cases to test new and modified functionality of software systems. We use the term *Test Reuse Pattern* to indicate common practices adopted by software developers to the activity of reusing and adapting test cases.

To identify *Test Reuse Patterns* applied by software developers to write new test cases from existing ones, we analyzed test cases belonging to open-source systems: JFreeChart[1], PMD[2], and JodaTime[3]. The goal of our analysis is to identify typical activities performed by developers when they reuse existing test cases to write new ones. In more detail, we identify pairs of similar test cases which help us to understand when software developers write similar test cases by reusing and adapting existing test cases. Our analysis identified both the test cases that the developers write by reusing existing test cases, and the test cases that the developers write from scratch. Our final goal was to identify general test reuse patterns.

We compare all the possible pairs of test methods in the projects using a tool called RefactoringCrawler [DCMJ06]. RefactoringCrawler has been originally designed to detect refactoring in evolving software. We modified its kernel to extract and display the differences of similar test method pairs in a comparison dialog box as the one is shown in Figure 3.2. We then manually identified the

[1]http://www.jfree.org
[2]http:/pmd.sourceforge.net
[3]http://jodatime.sourceforge.net

similarities and differences of test pairs.

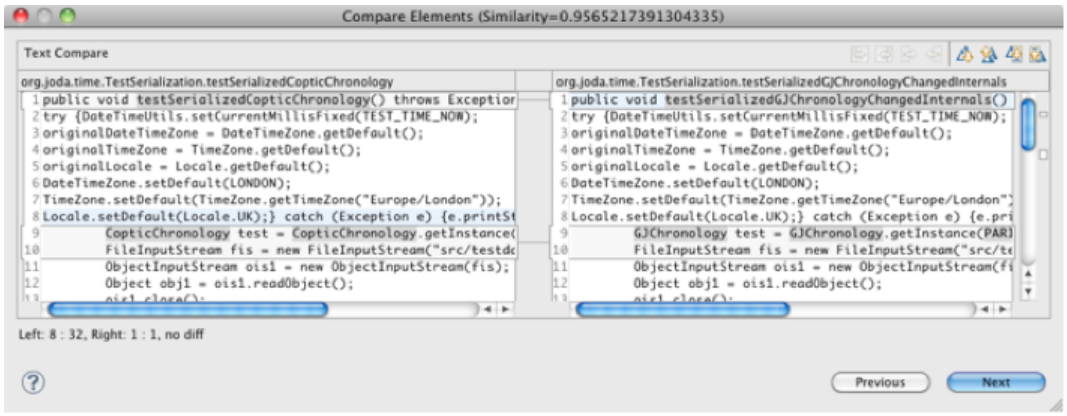

Figure 3.2. Comparing Similar Test Pairs in `CopticChronology` of JodaTime

We categorized the modified test suite as unchanged, removed, changed, and added. We used fully qualified names of methods to identify these categories. We then compared the changed and added test cases in modified software with original test cases. Figure 3.3 shows an abstract view of these categories.

By comparing the structure of test pairs we detected recurrent *Test Reuse Patterns* that are applied for different kinds of changes: method declaration changes, extension of class hierarchies, implementation of interfaces, introduction of new overloading methods, and introduction of overriding methods. In the following sections we describe each *Test Reuse Pattern*.

3.2.1 Change of Method Declaration

Software developers often change method declaration when they refactor a software system. As a consequence, they need to update existing test cases. The test cases that execute the modified method do not compile anymore and the developer need to fix the compilation error. Our experiments show that, apart from simple refactoring like rename of method, most of the changes that cause compilation errors are changes in the method declarations. The changes that cause compilation error in method declaration are:

- Parameter Type Change
- Parameter Addition
- Parameter Removal

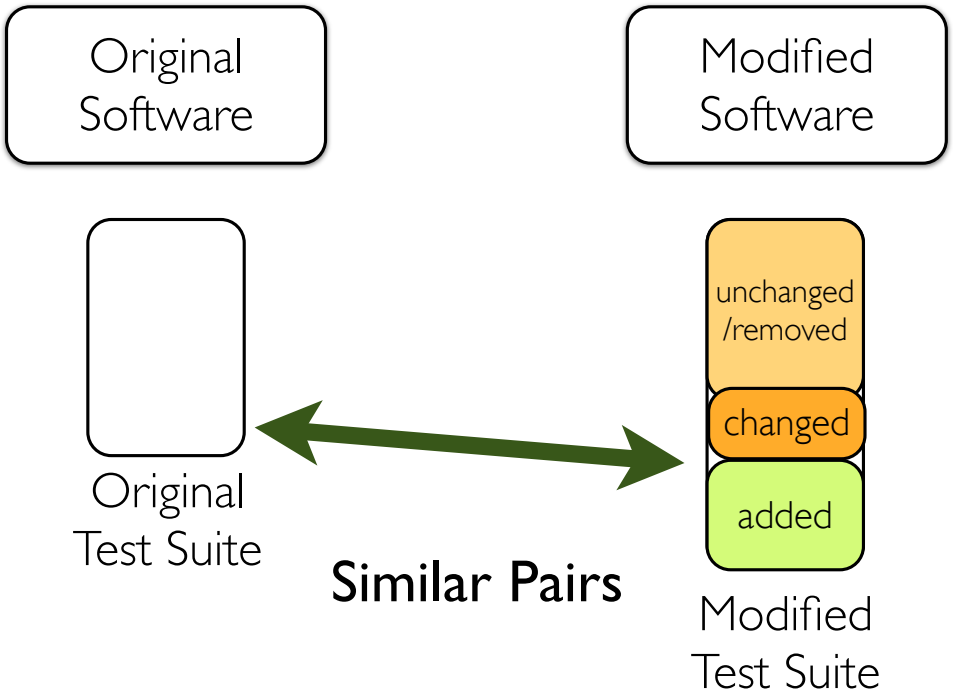

Figure 3.3. Similar Test Pairs in Empirical Studies

- Return Type Change

We illustrate this *Test Reuse Pattern* by considering an example[4] taken from PMD[5] a well known open-source program that checks static properties of Java applications.

Listings 3.1 and 3.2 show how PMD developers modified the declaration of method `Report.addRule` during the development of version 1.1 of PMD: they changed the type of the second parameter from `String` to `Context`.

One of the consequences of the change is the compilation errors in the 13 test cases that use the modified method. Listing 3.3 shows one of the broken test cases. The change in the method declaration caused a compilation error on line 4 because the test is passing variable `filename`, which is of type `String`, the second parameter of method `Report.addRule`, but method `Report.addRule` requires a parameter of type `Context` in version 1.1 of PMD.

Listing 3.4 shows the repair for the test case of Listing 3.3: the developers instantiated a new object of type `Context` and used the value of variable `filename` to initialize the new object `ctx` (line 3). This way PMD developers preserved the

[4]We slightly modified the example to make it easier to understand.
[5]http://pmd.sourceforge.net

behavior of the test; a random `String` or the empty `String` would not properly test the functionality of method because the name should correspond with the name of an existing file.

```
1 public void addRule(int line, String file){
2   this.line = line;
3   this.name = file;
4 }
```

Listing 3.1. Method Report.addRule in PMD 1.0

```
1 public void addRule(int line, Context ctx){
2   this.line = line;
3   this.name = ctx.getFilename();
4 }
```

Listing 3.2. Method Report.addRule in PMD 1.1

```
1 public void testBasic() {
2   Report r= new Report();
3   String filename="foo";
4   r.addRule(5, filename);
5   assertTrue(!r.isEmpty());
6 }
```

Listing 3.3. Test case for method `Report.addRule` in PMD 1.0

```
1 public void testBasic() {
2   Report r= new Report();
3   Context ctx= new Context("foo");
4   r.addRule(5, ctx);
5   assertTrue(!r.isEmpty());
6 }
```

Listing 3.4. Test case of Listing 3.3 repaired to work with PMD 1.1

Although test maintenance activities such as the change of PMD test case `testBasic` could be straightforward, the number of test cases to be repaired can be very large. Developers often do not have time to update all the test cases along with the modified classes, thus many test cases become obsolete. While software systems evolve, many obsolete test cases diverge from the system and become hard to repair, thus developers need to rewrite new test cases [BWK05]. The change of method `Report.addRule` is an example of a fairly common situation. Existing automated solutions such as automated refactoring techniques cannot handle this simple problem because file "foo" should exist.

IDEs like Eclipse provide simple solutions that might help software developers. In fact the refactoring menu of Eclipse provides the *Introduce Parameter Object* action that allows developers to automatically replace a parameter with a new type introduced by the developer [FBB+99]. This menu action automates the boxing of the object and automatically modifies also the test cases to generate a test like the one in Listing 3.4 thus preventing the compilation error.

Unfortunately the simple repair provided by Eclipse does not work in multiple cases:

- It is not applicable when a parameter should be replaced with an existing type;
- It does not allow developers to introduce types that require the invocation of a setter to encapsulate the value in the object (the only possible behavior is to pass the original value as argument of the constructor);
- It is not applicable to multiple methods (once applied to a method it generates the new type and thus cannot be applied to other methods);
- It is not applicable when developers need to manually modify the method declaration;
- It is not applicable when test cases are developed as side projects (this is a general limitation of many automated refactoring techniques).

The PMD case study described in this section is a typical example of an application that cannot be successfully maintained using simple techniques like the ones implemented by IDEs: the class `Context` has not been introduced with the refactoring of version 1.1 but was already used in version 1.0, furthermore the refactoring has been applied to two overloaded versions of `addRule`.

3.2.2 Extension of Class Hierarchy

In object oriented systems developers often extend software functionality by adding classes to hierarchies. Whenever a new class is introduced, developers write test cases to verify the functionality of the new class. To save testing effort, software developers reuse existing test cases and adapt test inputs and oracles to the specification of the new class. This practice is successful when the new class shares some behaviors with its siblings and parent classes. Therefore, existing test cases that verify sibling and parent classes can be successfully reused to test the new class. We discovered that the test cases designed

for similar siblings and sub-siblings can be reused as a template to derive the test cases for new class. We illustrate this *Test Reuse Pattern* with the example shown in Figure 3.4, which is a portion of class hierarchies in JodaTime version 1.2. To write test cases for class `EthiopicChronology`, test cases of classes `CopticChronology` and `IslamicChronology` can be reused as the candidate test cases (because classes `BasicFixedMonthChronology` and `BasicChronology` are abstract). This choice depends on the fact that all the classes in a hierarchy might share a common behavior thus their test cases can be used to write new test cases for the classes in the hierarchy. For example, developers use the test cases written for `CopticChronology` (Listing 3.6) to write test cases for class `EthiopicChronology`, shown in Listing 3.5. The structure of test cases is often similar except class references. Other than that, some input and oracle values are different because each Chronology employs specific algorithms to calculate date and time.

Software developers use their domain knowledge to identify the most similar siblings and sub-siblings of the class under test. Then, they copy the test case for that class, and finally modify the test inputs and outputs to fit the specifications of the new class.

```
1  public void testEpoch() {
2      ETHIOPIC_UTC = EthiopicChronology.getInstanceUTC();
3      JULIAN_UTC = JulianChronology.getInstanceUTC();
4      DateTime epoch = new DateTime(1, 1, 1, 0, 0, 0, 0,
           ETHIOPIC_UTC);
5      assertEquals(new DateTime(8, 8, 29, 0, 0, 0, 0, JULIAN_UTC),
           epoch.withChronology(JULIAN_UTC));
6  }
7
8  public void testEra() {
9      ETHIOPIC_UTC = EthiopicChronology.getInstanceUTC();
10     assertEquals(1, EthiopicChronology.EE);
11     try {
12         new DateTime(-1, 13, 5, 0, 0, 0, 0, ETHIOPIC_UTC);
13         fail();
14     } catch (IllegalArgumentException ex) {}
15 }
```

Listing 3.5. Two Test Methods in test suite of `EthiopicChronology`

```
1  public void testEpoch() {
2      COPTIC_UTC = CopticChronology.getInstanceUTC();
3      JULIAN_UTC = JulianChronology.getInstanceUTC();
```

```
4       DateTime epoch = new DateTime(1, 1, 1, 0, 0, 0, 0,
            COPTIC_UTC);
5       assertEquals(new DateTime(284, 8, 29, 0, 0, 0, 0, JULIAN_UTC
            ), epoch.withChronology(JULIAN_UTC));
6   }
7
8   public void testEra() {
9       COPTIC_UTC = CopticChronology.getInstanceUTC();
10      assertEquals(1, CopticChronology.AM);
11      try {
12          new DateTime(-1, 13, 5, 0, 0, 0, 0, COPTIC_UTC);
13          fail();
14      } catch (IllegalArgumentException ex) {}
15  }
```

Listing 3.6. Two Test Methods in test suite of `CopticChronology`

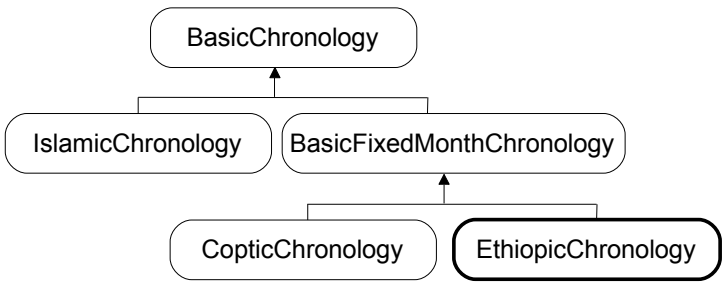

Figure 3.4. A Portion of Class Hierarchy in JodaTime Version 1.2

3.2.3 Implementation of Interface

The test cases developed to verify two classes that implement the same interface usually share a common behavior and differ only in terms of input values and oracles. Software developers often write test cases for new classes by reusing the test cases of classes that implement the same interfaces of new class. Since reused test cases are not originally designed for the new class, the test case might show some incompatibility problems such as invalid inputs. The developers use their domain knowledge to adapt inputs and oracles for new class and augment/remove some of the the test cases.

We illustrate this *Test Reuse Pattern* using an example taken from the open-source project JFreeChart version 1.013. Classes `FastScatterPlot` and `Compass Plot` implement the `Cloneable` and `Serializable` interfaces. The body of the

test cases for `FastScatterPlot` and `CompassPlot` is written by JFreeChart developers is the same except for some inputs and oracles. Listing 3.7 shows a test method written for `FastScatterPlot` and the corresponding test case for class `CompassPlot` is shown in Listing 3.8. The templates of these two classes are similar. The main difference is that the constructor of `CompassPlot` needs a parameter as input. Developers can identify the inputs and oracles that must be modified by replacing some values used in the original test cases with values taken from the specification of the new class. This *Test Reuse Pattern* is often performed by developers to reuse and adapt existing test cases to write new test cases instead of writing them from scratch.

```
1  public void testCloning() {
2      FastScatterPlot p1 = new FastScatterPlot();
3      FastScatterPlot p2 = null;
4      try {
5          p2 = (FastScatterPlot) p1.clone();
6      }
7      catch (CloneNotSupportedException e) {
8          e.printStackTrace();
9      }
10     assertTrue(p1 != p2);
11     assertTrue(p1.getClass() == p2.getClass());
12     assertTrue(p1.equals(p2));
13 }
```

Listing 3.7. A Test Method in test suite of `FastScatterPlot`

```
1  public void testCloning() {
2      CompassPlot p1 = new CompassPlot(new DefaultValueDataset
           (15.0));
3      CompassPlot p2 = null;
4      try {
5          p2 = (CompassPlot) p1.clone();
6      }
7      catch (CloneNotSupportedException e) {
8          e.printStackTrace();
9      }
10     assertTrue(p1 != p2);
11     assertTrue(p1.getClass() == p2.getClass());
12     assertTrue(p1.equals(p2));
13 }
```

Listing 3.8. A Test Method in test suite of `CompassPlot`

3.2.4 Introduction of Overloaded Method

The overloaded methods of a class share their names, but present different number and/or type of parameters. Software developers reuse the test cases developed for an overloaded method, usually the first one developed, to write the test cases for the other overloaded methods by modifying the input parameters according to the signature[6] of the new method, and by altering the oracles according to the implementation of the new method.

We illustrate this *Test Reuse Pattern* using an example taken from the project JodaTime version 1.1. The method `getInstance()` in class `BuddhistChronology`, has been overloaded by the method `getInstance(DateTimeZone)`. Listing 3.9 shows the test case written for method `getInstance()`, while Listing 3.10 shows the test method written for overloaded method. As the two test methods suggest, the developer used the same template to write the test case for the overloading method. The test cases written for the overloading method match the test cases for the overloaded one except for the inputs of the parameters and few oracles. There are some new method calls are added to the test case for overloaded method to exercise different values for new parameter `DateTimeZone`.

```
1 public void testFactory() {
2     assertEquals(LONDON, BuddhistChronology.getInstance().
          getZone());
3     assertSame(BuddhistChronology.class, BuddhistChronology.
          getInstance().getClass());
4 }
```

Listing 3.9. Test Method calling `getInstance()`

```
1 public void testFactory_Zone() {
2     assertEquals(TOKYO, BuddhistChronology.getInstance(TOKYO).
          getZone());
3     assertEquals(PARIS, BuddhistChronology.getInstance(PARIS).
          getZone());
4     assertEquals(LONDON, BuddhistChronology.getInstance(null).
          getZone());
5     assertSame(BuddhistChronology.class, BuddhistChronology.
          getInstance(TOKYO).getClass());
6 }
```

Listing 3.10. Test Method calling `getInstance(DateTimeZone)`

[6]Method Signature is defined by the name, plus the number and the type of parameters

3.2.5 Introduction of Overridden Method

When developers override a method, they write a new method with the same signature of a method defined in a parent class. Software developers reuse the test cases developed for the method of the parent class to test the new method: they change the oracles to verify the expected behavior of the child class, for example by inspecting the value of a field modified in the overridden method, and alter the method inputs if the input domain of the overloading method has been changed.

```
1 public void testHashcode() {
2    CategoryTextAnnotation a1 = new CategoryTextAnnotation("Test"
        ,"Category", 1.0);
3    CategoryTextAnnotation a2 = new CategoryTextAnnotation("Test"
        ,"Category", 1.0);
4    assertTrue(a1.equals(a2));
5    int h1 = a1.hashCode();
6    int h2 = a2.hashCode();
7    assertEquals(h1, h2);
8 }
```

Listing 3.11. Test Method of class `CategoryTextAnnotation`

```
1 public void testHashcode() {
2    CategoryPointerAnnotation a1 = new CategoryPointerAnnotation(
        "Label","A", 20.0, Math.PI);
3    CategoryPointerAnnotation a2 = new CategoryPointerAnnotation(
        "Label","A", 20.0, Math.PI);
4    assertTrue(a1.equals(a2));
5    int h1 = a1.hashCode();
6    int h2 = a2.hashCode();
7    assertEquals(h1, h2);
8 }
```

Listing 3.12. Test Method of class `CategoryPointerAnnotation`

We illustrate this pattern with an example from the JFreechart project version 1.013: The method `equals()` is overridden in class `CategoryPointerAnnotation` that extends class `CategoryTextAnnotation`. The test case written for method `equals()` in class `CategoryTextAnnotation`, shown in Listing 3.11, can be reused to test the overridden method in class `CategoryPointerAnnotation`, shown in Listing 3.12. The test cases differ only in the input variables of constructor calls of the classes. The structure of the test cases written for the overridden method is the same as base method except for some input values. The

developers usually copy the test cases from base method and using their application knowledge replace the new input values with old ones.

This chapter provides an overview of our framework which repairs not compiling test cases and adapts test cases for new version of the software. Our analysis of empirical data collected from open-source projects, suggests that the developers reuse existing test cases to test modified software and generate new ones. We call these recurrent activities by the developers as *Test Reuse Pattern*. We identify five *Test Reuse Patterns* including change of method declaration, extension of class hierarchy, implementation of interface, introduction of overloaded method, and introduction of overridden method. We discuss each *Test Reuse Pattern* with an example taken from test case repository of open-source projects.

Chapter 4

Test Suite Repair

This chapter details the algorithm that automates the "Change of method declaration" *Test Reuse Pattern*. This *Test Reuse Pattern* is triggered by the developers who observe some compilation errors in the test cases while modifying method declarations in the software. TCA proposes a possible fix for the test case by applying the algorithm described in this section. The modified test case can be validated by the developer and submitted to the repository. We refer to this adaptation process as test repair since we remove compilation error when we modify the test case.

We first present a simple motivating example then we describe the algorithm by applying it step by step on the motivating example. To repair the test case that does not compile correctly due to some method modifications, TCA first identifies the type of changes that caused compilation error in method invocations. The types of changes can be parameter type change, parameter removal, parameter addition, and return type change. According to the type of change, TCA follows a slightly different approach to repair the test case. In all types of changes,TCA extracts the usage of the modified parameters and finds the corresponding changes in the usage locations. Then TCA identifies definition-use pairs to find the values that preserve the behavior of the test cases. Finally, TCA applies the changes to test cases and removes the compilation errors.

4.1 Motivating Example

In this section, we illustrate the kinds of changes that TCA can address with a simple bank account management system which is used in literature [JOX10a, JOX10b]. Figures 4.1 to 4.4 show four types of changes and illustrate how developers repair the associated test cases to solve compilation errors.

Figure 4.1 shows a change of parameter type: the type of the first parameter of method `deposit` has changed from `int` to `Money`. In fact, the parameter `cent` is encapsulated in the type `Money`. This change does not modify the functionality of the method but is applied to refactor the method to improve its readability and maintainability. In this case, the existing automated refactoring techniques to repair test cases suffer from one or more of the following problems:

- They are not applicable when the parameter `cents` should be replaced with an existing type `Money`;
- They do not allow developers to introduce type `Money` that requires the invocation of a setter to encapsulate the `cents` value in the object;
- They are not applicable to multiple methods (once applied to a method it generates the new type and thus cannot be applied to other methods);
- They are not applicable when developers need to manually modify the method declaration;
- They are not applicable when test cases are developed as side projects (this is a general limitation of many automated refactoring techniques).

```
1 public class BankAccount {
2   private int balance;
3
4   public void deposit(int cents, String currency){
5     balance += cents * getChange(currency);
6   }
7 }
```

Figure 4.1.1: Original Software

```
1 public class BankAccount {
2   private int balance;
3
4   public void deposit(Money m, String currency){
5     balance += m.centsValues * getChange(currency);
6   }
7 }
```

Figure 4.1.2: Modified Software

```
1 BankAccount account = new BankAccount();
2 int amount = 500;
3 account.deposit(amount,"EUR");
4 assertEquals( 500, account.getBalance());
```

Figure 4.1.3: Original Test Case

```
1 BankAccount account = new BankAccount();
2 Money amount = Money(500);
3 account.deposit(amount,"EUR");
4 assertEquals( 500, account.getBalance());
```

Figure 4.1.4: Repaired Test Case

Figure 4.1. Parameter Type Change

This change causes a compilation error in line 3 of the original test case (Figure 4.1.3). The original test case invokes method `deposit` by passing an integer parameter, while the new test case must pass an object of type `Money` (line 3). Developers would repair the test case in Figure 4.1.3 by replacing the `int` variable `amount` passed to method `deposit` with an object of type `Money`

```
1 public class BankAccount {
2   private int balance;
3   private double dailyInterestRate = 0.00005;
4   private int interestTerm=365;
5   public double interest(){
6     return balance*dailyInterestRate*interestTerm;
7 }}
```

Figure 4.2.1: Original Software

```
1 public class BankAccount {
2   private int balance;
3   private double dailyInterestRate = 0.00005;
4
5   public double interest(int days){
6     return balance*dailyInterestRate*days;
7 }}
```

Figure 4.2.2: Modified Software

```
1 BankAccount account = new BankAccount();
2 account.deposit(500);
3 int interestEarned = account.interest();
4 assertEquals(9.125,interestEarned);
```

Figure 4.2.3: Original Test Case

```
1 BankAccount account = new BankAccount();
2 account.deposit(500);
3 int days = 365;
4 int interestEarned = account.interest(days);
5 assertEquals(9.125,interestEarned);
```

Figure: 4.2.4: Repaired Test Case

Figure 4.2. Parameter Add Example

(Figure 4.1.4). To preserve the test behavior, that is to obtain the same results of the original software, developers must initialize the new object of type `Money` with value `500`.

Figure 4.2 shows the case of adding a parameter to method `interest`. Parameter `days` of type `int` is added to method `interest`. This change is highlighted in line 6 of Figures 4.2.1 and 4.2.2, and causes a compilation error in line 3 of the original test case (Figure 4.2.3). This change is not only a simple parameter addition but also a change in the structure of the class. In fact, the field `interestTerm` is moved to the parameter of method `interest`. The developer would like to have more flexibility in calculating the interest of a certain account by dynamically assigning values to the number of days in which interest should be calculated. This change in the method declaration cannot be fully applied by using the automated "introduce parameter" refactoring tools because the refactoring raises compilation errors in many test cases due to the new interface. The developers need to go through each compilation error in their test workspace and manually fix the errors. Developers would repair the test cases by passing value `365` to method `interest` because `365` is the value of the variable `interestTerm` used in the original software (Figure 4.2.4).

Figure 4.3 shows an example of parameter removal. The parameter `currency` of method `deposit` is removed, and the parameter `currency` is encapsulated in the objects of type `Money` in the modified software as shown in Figure 4.3.2. In fact, the parameter `currency` is wrapped into the class `Money`, and this results in a compilation error in the test case of Figure 4.3.3. To repair the test case and preserve its behavior, developers would remove the parameter from the call of method `deposit` in Line 4 of test case (Figure 4.3.3), and initialize the field

```
1 public class BankAccount {
2   private int balance;
3
4   public void deposit(Money mon, String currency){
5     balance += mon.cents * change(currency);
6   }
7 }
```

Figure 4.3.1: Original Software

```
1 public class BankAccount {
2   private int balance;
3
4   public void deposit(Money mon){
5     balance+=mon.cents*change(mon.getCurrency());
6   }
7 }
```

Figure 4.3.2: Modified Software

```
1 BankAccount account = new BankAccount();
2 Money amount = new Money(500);
3
4 account.deposit(amount,"EUR");
5 assertEquals( 500, account.getBalance());
```

Figure 4.3.3: Original TestCase

```
1 BankAccount account = new BankAccount();
2 Money amount = new Money(500);
3 amount.setCurrency("EUR");
4 account.deposit(amount);
5 assertEquals( 500, account.getBalance());
```

Figure 4.3.4: Repaired Test Case

Figure 4.3. Parameter Remove Example

```
1 public class BankAccount {
2   private int balance;
3   public void deposit(int cents){
4     balance += cents;
5   }
6   public int getBalance(){
7     return balance;
8 }}
```

Figure 4.4.1: Original Software

```
1 public class BankAccount {
2   private int balance;
3   public void deposit(int cents){
4     balance += cents;
5   }
6   public Money getBalance() {
7     Money mon = new Money( balance );
8     return mon;
9 }}
```

Figure 4.4.2: Modified Software

```
1 BankAccount account = new BankAccount();
2 account.deposit(500);
3 int balance =account.getBalance();
4 assertEquals( 500, balance );
```

Figure 4.4.3: Original Test Case

```
1 BankAccount account = new BankAccount();
2 account.deposit(amount);
3 Money money =account.getBalance();
4 assertEquals( 500, money.getCentsValue() );
```

Figure 4.4.4: Repaired Test Case

Figure 4.4. Return Type Change Example

`Money.currency` to `EUR`, the value of the removed argument (Figure 4.3.4). The field `currency` of class `Money` is initialized by calling the corresponding setter (Line 3 of Figure 4.3.4).

Finally, Figure 4.4 shows an example of change in the return type. The change causes a compilation error in line 3 of the test case in Figure 4.4.3. The return type of method `getBalance` is changed from `int` to `Money`. The object of type `Money` returned by the method encapsulates the value `int` returned by the original software. To fix the compilation error, developers would substitute the result of method `getBalance` with an invocation of method `getCentsValue` on the object of type `Money` returned by `getBalance`. Method `getCentsValue` returns the `int` representation of the money amount in the bank account, as shown in Figure 4.4.4.

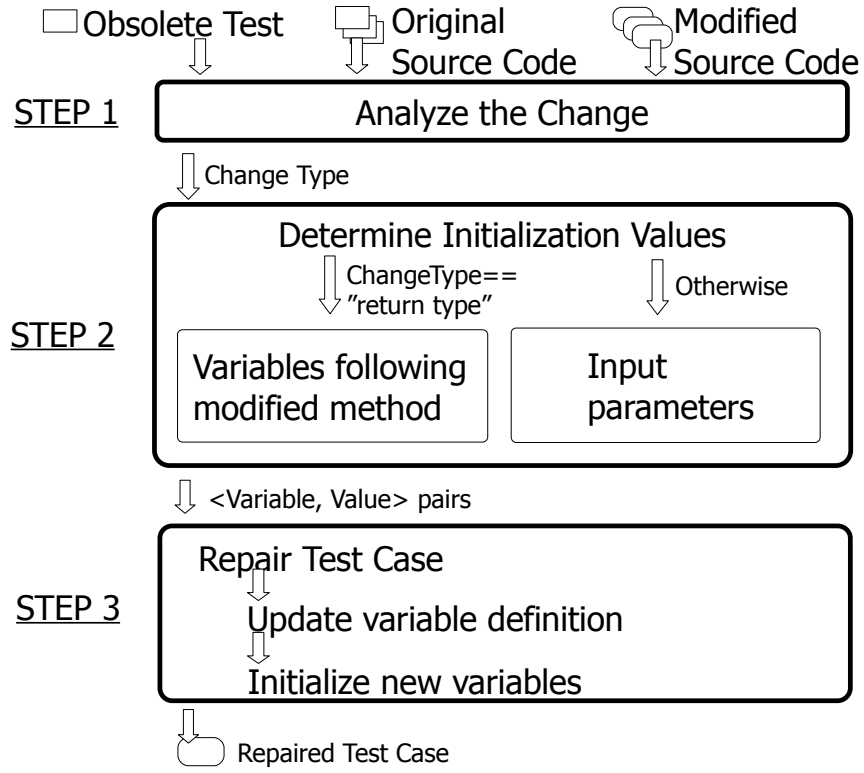

Figure 4.5. The TCA approach to repair test cases

4.2 Automatic Test Repair

TCA focuses on compilation errors that derive from changes in the declaration of methods that occur when developers modify the type of the value returned by the method, and the number or type of the input parameters of the method. TCA does not target simple compilation errors caused by method renaming or changes in the parameters order, because developers usually perform these modifications using refactoring tools that prevent compilation errors. TCA also ignores modifications that do not raise any compilation errors in test cases, for example the introduction of a return type i.e., changing the return type from "void" to some actual type.

Figure 4.5 shows the three main activities that TCA iterates to repair the compilation errors: analyze the change, determine the initialization values, and repair the test case.

TCA relies on the static def-use analysis of the class under test and their dependent entities, and uses dynamic instrumentation and execution to get runtime values of certain variables. The complexity of the algorithms presented in

this section is of the same order of the complexity of the employed static and dynamic analysis techniques (polynomial in the worst case).

The next three sections describe the activities in detail.

4.3 Analyze the Change

TCA starts by identifying both the changed elements (the modified parameters and return values) and the type of change. TCA first identifies the methods that need to be repaired using the information about the compilation errors. If the compilation errors do not involve method invocations, TCA cannot fix the errors and terminates. Otherwise, TCA identifies the type of changes by simply diffing the declarations of the modified methods. In the current prototype, we target Java programs, and we use JDiff[1]. From the differences between the methods before and after the changes, TCA identifies the set of elementary actions that comprise the changes in terms of parameter additions, removals, type changes, and return type changes. For instance, with the example of *parameter type change* in Figure 4.1 as input, TCA determines that the type of the first input parameter of method `deposit` has been changed from `int` to `Money`.

TCA proceeds with the test repair step if JDiff detects on of these type of changes: return type change, parameter type change, parameter removal or parameter add. TCA does not work in the presence of generics or reflection, due to the limitations of the analysis techniques.

4.4 Determine the Initialization Values

After having identified the elementary change responsible for the compilation error, TCA determines both the program variables that must be initialized to preserve the behavior of the test case, and the proper initialization values. TCA determines the program variables to initialize by comparing static data flow information of the original and the modified software, and identifies the proper initialization values by dynamically analyzing the original software. The combination of static data flow and dynamic analysis produces a set of pairs $\langle variable, value \rangle$ that indicate the proper initialization values for the variables.

For instance, the analysis of the example of *parameter type change* in Figure 4.1 produces the pair $\langle money.centsValues, 500 \rangle$ that indicates that the field

[1] http://www.jdiff.org/

`centsValues` of the parameter `Money` in the method `deposit` should be initialized with `500` to preserve the behavior of the test case.

TCA adopts different analysis strategies according to the type of change identified in the first phase as it is shown in step 2 of Figure 4.5:

Input parameters: If the change involves input parameters (i.e., parameter type change, parameter addition, parameter removal), the variables to be fixed are declared before the method invocation. In fact, to resolve the compilation error, TCA modifies the variable assigned right before the method call in the test case. This process is described in Section 4.4.1.

Variables following modified method: If the change involves the return type of a method (return type change), TCA modifies the return variable of the modified method, and all the usages of the returned variable that appear after the method invocation. This process is described in Section 4.4.2.

4.4.1 Input Parameters

TCA identifies the initialization values for the input parameters with the algorithm $initialize_{pre}$ presented in Figure 4.6. The algorithm identifies the variables to initialize, $VarsToInit_{pre}$, and their initialization values, in two different ways according to the type of change (line 1).

In the cases of parameter addition or parameter type change, TCA first identifies the variables to initialize by inspecting the def-use chain of the modified parameter, then TCA repairs the test by identifying the corresponding terms in the original software that hold the initialization values (lines 2–15). In case of parameter removal, TCA first identifies the removed input that was used in the test case, then identifies the corresponding values in the original software that should be initialized to preserve the behavior of the method (lines 17–28).

In both cases the algorithm $initialize_{pre}$ uses the functions $occurrencesOf$, $corresponding$, and $valueOf$. We first describe these three utility functions that are required to explain the $initialize_{pre}$ algorithm.

Function $occurrencesOf(v, S)$: computes the set of uses of variable v, including the uses induced by the inspector methods in Software S. The function $occurrencesOf(v, S)$ is based on classic interprocedural data flow analysis [HS94] and is not further described in this dissertation.

Require: CT change type
Require: CE changed element (parameter)
Require: S_0 original software
Require: T_0 test to repair
Require: S_1 modified software
Ensure: a list of pairs $\langle v1, value \rangle$

if $CT \in \{Add, TypeChange\}$ **then**
  $VarsToInit_{pre} \leftarrow identifyVarsUsed(CE, S_1)$
  for all $v1 := VarsToInit_{pre}$ **do**
    $Occs_{v1} \leftarrow occurrencesOf(v_1, S_1)$
    $i \leftarrow 0$
    while $occ_{v0} = \perp$ **and** $i \leq \|Occs_{v1}\|$ **do**
      $occ_{v1} \leftarrow Occs_{v1}[i]$
      $occ_{v0} \leftarrow corresponding(S_1, S_0, occ_{v1})$
      if $occ_{v0} \neq \perp$ **then**
        $val_{v0} \leftarrow valueOf(S_0, occ_{v0})$
      end if
      $i \leftarrow i + 1$
    end while
    $results \leftarrow results + \langle v1, val_{v0} \rangle$
  end for
else
  $TermsUsed_{pre} \leftarrow identifyVarsUsed(CE, S_0)$
  for all $v0 := TermsUsed_{pre}$ **do**
    $Occs_{v0} \leftarrow occurrencesOf(v_0, S_0)$
    $i \leftarrow 0$
    while $occ_{v1} = \perp$ **and** $i \leq \|Occs_{v0}\|$ **do**
      $occ_{v0} \leftarrow Occs_{v0}[i]$
      $occ_{v1} \leftarrow corresponding(S_0, S_1, occ_{v0})$
      $val_{v0} \leftarrow valueOf(S_0, occ_{v0})$
      $i \leftarrow i + 1$
    end while
    $results \leftarrow \langle v1, val_{v0} \rangle$
  end for
end if

Figure 4.6. The $inititialize_{pre}$ algorithm.

Function $corresponding$: maps the occurrence of a variable in one version of the software to the occurrence of corresponding variable in another version of the software. Function $corresponding(S_x, S_y, occ_{ax})$ receives as input the occurrence occ_{ax} of a term a in the version S_x of the software, and returns the occurrence occ_{by} of the term b that corresponds to occ_{ax} in the version S_y. The occurrence occ_{by} substitutes occ_{ax} in the operation modified from the original to the modified version of the software. For example, in Line 5 of Figure 4.1, variable `m.centsValues` corresponds to `cents`. TCA prototype implements the function $corresponding$ by combining the *Unix diff algorithm*, the Levenshtein distance [Lev65], and the global strings alignment algorithm [NW70]. It applies the *Unix diff algorithm* to the two versions S_x and S_y of the method that contains occ_a to identify a line l_y in the version S_y that corresponds to the line l_x that contains occ_{ax} in the version S_x. If the *Unix diff* identifies multiple counterparts for line l_x, TCA selects the line most similar to l_x according to the Levenshtein distance. Given l_x and l_y, TCA identifies occ_{ax} by applying the global alignment algorithm to l_x and l_y and finds the term that corresponds to occ_{ax}.

Function $valueOf$: computes the value assigned to a variable at runtime in a specific version of software. Function $valueOf(S_0, occ_{a0})$ combines dynamic tracing and interprocedural data flow analysis to compute the value assigned to the occurrence occ_{a0} of a variable in the original version S_0 of the software. TCA traces the runtime values of a variable by executing an instrumented version of the original software. If the monitored variable is a primitive variable, the trace records all the required information, if the variable is an object (non-primitive type), TCA recursively visits the object graph and records the values of the primitive fields of object. In both cases, TCA uses the recorded values to initialize the variable. This approach works well in most of the cases, but not when the test case compares references. In the presence of references, TCA first tries to statically determine if there is an alias within the test case. If TCA finds the alias, TCA uses the alias to initialize the target variable, otherwise it combines dynamic tracing with interprocedural data flow analysis to identify a suitable value for initializing the target variable.

We now discuss the case of parameter addition or parameter type change (lines 2–15). TCA first identifies the set $VarsToInit_{pre}$ of the variables to be initialized by invoking the utility function $identifyVarsUsed()$ with the changed element, CE, and the modified version of the code (S_2 in line 2). If the changed element (a

modified parameter, in this case) is primitive, $identifyVarsUsed()$ returns the parameter itself, if the modified parameter is an object, $identifyVarsUsed()$ returns the fields used within the modified method. We identify single fields because setting the state of an object may imply initializing different object fields through the invocation of several methods, and TCA should consider only the object attributes used in the modified method. $identifyVarsUsed()$ uses the *DaTeC* data flow analyzer [GAM09] to identify all the uses of the modified parameter fields, and the *Soot* static analyzer [VRCG+99] to filter out the fields not accessed within the call graph of the modified method. Soot data flow analyzer accepts fully qualified name of class under test and performs static data flow analysis on class under test and all dependent classes.

For each variable in the set $VarsToInit_{pre}$, TCA identifies the first occurrence that has a corresponding occurrence in the original version of the software, and detects the runtime value of the occurrence in the original version (lines 3–15). The main cycle (lines 3–15) loops on the variables in the set, the inner cycle (lines 6–13) loops on the occurrences of each variable until it either finds a correspondence in the original software ($occ_{v0} \neq \perp$) or there are no correspondences at all ($occ_{v0} = \perp \wedge i > \|occs_{v1}\|$). TCA uses function $valueOf()$ to identify the runtime value of occ_{v0} (line 10).

In the case of parameter removal (lines 17–28), TCA first derives the set of variables used in the original method, $TermsUsed_{pre}$, that contains the values to use for the repair. To this end, TCA invokes the function $identifyVarsUsed()$ with the changed element and the original version of the code. Then, for each variable in the set $TermsUsed_{pre}$, TCA identifies the first occurrence that has a corresponding occurrence in the modified version of the software, and detects the runtime value of the occurrence in the original version (lines 18–28). The main cycle loops on the variables in the set $TermsUsed_{pre}$ (lines 18–28), the inner cycle loops on the occurrences of each variable until it either finds a correspondence in the modified software ($occ_{v1} \neq \perp$) or there are no correspondences at all ($occ_{v1} = \perp \wedge i > \|occs_{v1}\|$) (lines 21–26). TCA uses function *valueOf()* to identify the runtime value of occ_{v0} (line 27).

4.4.2 Variables Following Modified Method

When the modifications of the method affect the type of the returned value, TCA needs to change definition of the returned value of the modified method with the new type. Moreover, TCA should adapt the uses of the value returned by the modified method. An example is given in Figure 4.4: The modified version of method `getBalance()` does not return the balance as an integer value, but as an

object of type `Money` (Line 6 in Figure 4.4.2) that contains the balance as a primitive field. In the modified version, method `Money.getCentsValue()` returns the integer representation of the `balance`, i.e., the same value returned by method `BankAccount.getBalance` in the original version of the software. TCA finds method `Money.getCentsValue()` by extracting the methods that define field `balance` using DaTeC. TCA repairs the test case of Figure 4.4 by defining a new variable of modified returned type (`mon`) and initializing the second parameter of `assertEquals` in line 6 with the value returned by method `getCentsValue()` invoked on the object `mon` returned by the modified method.

Require: CE changed element (return type)
Require: S_0 original software
Require: T_0 test to repair
Require: S_1 modified software
Ensure: a pair $\langle varToInit, inspectorSequence \rangle$
1: $varToInit \leftarrow identifyVarToInit(S_0, T_0)$
2: $retTerm \leftarrow identifyReturnedVar(S_0, varToInit)$
3: $RetTermDefs \leftarrow definitions(S_0, retTerm)$
4: $i \leftarrow 0$
5: **while** $occ_{v1} = \perp$ **and** $i \leq \|RetTermDefs\|$ **do**
6: $occ_{v0} \leftarrow RetTermDefs[i]$
7: $occ_{v1} \leftarrow corresponding(S_0, S_1, occ_{v0})$
8: **if** $occ_{v0} \neq \perp$ **then**
9: $occ_{v1} \leftarrow identifyInspector(T_0, S_1, occ_{v0})$
10: **end if**
11: $i \leftarrow i + 1$
12: **end while**
13: **return** $\langle varToInit, occ_{v1} \rangle$

Figure 4.7. The $inititialize_{post}$ algorithm.

TCA repairs the test case of Figure 4.4 in two steps: First it declares a new variable, `money` in this case, of the same type of the new returned value. Then, it replaces all the uses of the value originally returned by the modified method with the invocation of `Money.getCentsValue`. In practice it replaces the occurrences of variable `balance` with `money.getCentsValue`. In this case the second parameter of `assertEquals` in line 6 with the value returned by method `getCentsValue()` invoked on the object `money` returned by the modified method.

Figure 4.7 shows the algorithm $initialize_{post}$ which identifies the initialization values for the variables used after invocation of the modified method. The

algorithm $initialize_{post}$ identifies the variable $varToInit$ initialized with the invocation of the modified method, and updates any usage of the returned variable in the test case. For example, in Figure 4.4, $initialize_{post}$ identifies `balance` as the variable initialized by the invocation of the modified method, then determines that `Money` is an encapsulator of variable `balance` and replaces an instance of `Money` with definition of `balance`. Moreover, $initialize_{post}$ replaces any usages of `balance` with an appropriate variable in the modified test case (in this example $initialize_{post}$ only finds one usage of parameter `balance` in Line 4 of Figure 4.4.3).

The algorithm $initialize_{post}$ first identifies $varToInit$ by invoking the function $identifyVarToInit()$ (line 1). The variable $varToInit$ is simply the variable defined with the value returned by the modified method. The algorithm $initialize_{post}$ then invokes function $identifyReturnedVar()$ to identify the program variable or constant, $retTerm$, returned by the modified method (line 2). Function $identifyReturnedVar()$ identifies $retTerm$ by executing an instrumented version of the program that keeps track of the return instruction executed during the invocation of the modified method.

The algorithm $initialize_{post}$ then identifies the set $RetTermDefs$ that contains the definition of variables with the same runtime value of $retTerm$ (line 3). The set $RetTermDefs$ contains both the definition def_t of the variable $retTerm$ and the definitions of those variables that are simple copies of def_t, if any.

Finally $initialize_{post}$ scans the set $RetTermDefs$ following the order of occurrence of the elements in the call tree of the modified method (lines 5 to 12). For each variable occurrence occ_{v0} that belongs to $RetTermDefs$, $initialize_{post}$ invokes function $corresponding()$ to identify occ_{v1}, the occurrence of a variable that corresponds to occ_{v0}. At runtime occ_{v1} contains the same value of $retTerm$. The algorithm $initialize_{post}$ then invokes the function $identifyInspector()$ to find the sequence of getter methods or fields that permits to retrieve the value of occ_{v1} in the test. Function $identifyInspector()$ analyzes the object graph of all the objects defined in the test case to determine if such sequence exists. The algorithm $initialize_{post}$ iterates these operations till it finds a variable that can be accessed within the test case, or until all the variable occurrences in $RetTermDefs$ have been inspected.

When applied to the return type example of Figure 4.4, $initialize_{post}$ identifies the return type change of method `getBalance`. The algorithm $initialize_{post}$ determines return type change from `int` to `Money`. The algorithm $initialize_{post}$ then determines that it should change the definition of returned variable from `int` to `Money` and assign value `balance` to $varToInit$. Moreover, the algorithm identifies all the locations of usage of the variable returned by modified

method. In this case, $initialize_{post}$ determined only on usage location of variable `balance` in the second argument of method `assertEquals` (Line 4 of Figure 4.4.3) and assign it to the variable occ_{v0} in the algorithm. The variable `balance` has a corresponding variable in the modified software, `balance` in line 2 of Figure 4.4.2 (variable occ_{v1} in Line 7). Since the variable `balance` is encapsulated in the class `Money`, $initialize_{post}$ looks for the corresponding field in class `Money`. The corresponding field is field `Money.centsValue`, which can be defined by constructor and accessed directly. Therefore, $initialize_{post}$ returns pair $\langle balance, mon.centsValue \rangle$ to be used by next step which applies fixes to the test case.

4.5 Repair Test Case

In last phase, TCA repairs the test case by first updating the variable definitions to fix the compilation errors, then by initializing the variables with proper values. The next paragraphs describe these activities in details.

4.5.1 Update Variable Definitions

The variable definitions are updated depending on the kind of change that causes the compilation error. In the case of parameter addition, TCA defines a new variable of the same type as the new parameter and modifies the method call by adding the new value for the new argument. In the example of *parameter add* in Figure 4.2, TCA repairs the compilation error by defining the new variable `days` of the same type of the introduced parameter (`int`), and using the variable `days` as argument of the modified method.

In the case of parameter type change, TCA defines a new variable of the same type of the modified parameter and passes it to the modified method in place of the original variable. In the example of *parameter type change* in Figure 4.1, TCA defines a new variable, `amount`, of type `Money`, the type of the modified parameter in the modified version, and passes it as the first parameter of method `deposit`. TCA also removes the declaration of `amount`, the variable passed in the original version, because it is not used by other method calls.

In the case of parameter removal, TCA modifies the method call by removing the argument that corresponds to the removed parameter. For example in the example of *parameter removal* in Figure 4.3, TCA removes the second argument of method deposit, the literal `EUR`.

In the case of return type change, TCA defines a new variable of the same

type of the new return type, and assigns it the value returned by the modified method. If the new return type is `void`, TCA does not define any variable, and replaces the original invocation of the modified method with a new variable.

4.5.2 Initialize New Variables

TCA initializes the introduced or modified variables in the test case in two different ways. In the case of parameter additions, parameters removal, and parameter type change, TCA changes the initialization of the variables in $VarsToInit_{pre}$before the invocation of the modified method. In the case of return type change TCA changes the initialization of the variable initialized with the value returned by the modified method.

The variables in $VarsToInit_{pre}$ can be primitive arguments of the method, or fields belonging to objects declared in the test case. TCA initializes the primitive arguments by simply assigning the values as computed in the phase "Determine the initialization Values" by the algorithm $initialize_{pre}$. For instance in the example *parameter add*, TCA assigns the value `365` to the variable `days`, the new argument of the modified method.

If the fields to initialize belong to objects declared in the test case, TCA distinguishes between objects initialized in the original test and objects added by TCA that need to be initialized. If the fields to initialize belong to an object already defined in the test case, TCA initializes the public fields by assigning the values computed by $initialize_{pre}$ and the private ones by invoking the corresponding setter methods. If a field does not have an associated setter method, TCA initializes the field by means of reflection [FF04].

Let us consider the problem of initializing the field `Money.currency` in the example of *parameter removal* in Figure 4.3.3. The object `money` passed to the method `deposit` is already defined before the invocation of method `deposit`, and is referenced by variable `amount`. TCA initializes the field `amount.currency` by invoking method `amount.setCurrency` (Figure 4.3.4). TCA uses the literal `EUR` as argument of method `setCurrency` because this is the value returned by $initialize_{pre}$.

If the fields to initialize belong to an object declared by TCA, TCA first instantiates the object by using the constructor that initializes most of these fields, then initializes the remaining fields as described in the previous paragraph. TCA sets the value of each parameter p of the constructor, as follow. If p initializes a field f that belongs to $VarsToInit_{pre}$, TCA uses the value returned by function $initialize_{pre}$ as the value to be passed as an argument. If p is not used to initialize a term in $VarsToInit_{pre}$, or if the function $initialize_{pre}$ is not defined

for field f, TCA uses a default value[2]. The example *parameter type change* in Figure 4.1 shows how TCA initializes a field of a newly introduced object. In this example TCA must initialize `money.centsValue`, a field of the first parameter of the modified method. Since the variable `money` has not been initialized in the test case, TCA identifies the constructor `Money(int value)` to initialize the field `centsValue`.

In the case of changes in the return type of a method m, TCA initializes the variable that uses the value returned by m with the value identified by function $initialize_{post}$. For instance, the test case of the example *return type change* in Figure 4.4 does not compile after the change of the return type of method `getBalance`. Figure 4.4.4 shows the repair performed by TCA that defines the new variable `money` of type `Money`, the type returned by the modified method, and assigns the value returned by the modified method to the variable `money`. Then TCA initializes the second argument of `assertEquals` in line 4 of Figure 4.4.4 with the value `money.getCentsValue()` returned by $initialize_{post}$.

[2]In the current prototype, TCA uses the following default values: 0 for numeric types, the empty string for strings, and null for objects.

Chapter 5

Test Suite Adaptation

This chapter details the algorithms that automate the *Test Reuse Patterns* for testing new functionality: extension of class hierarchies, implementation of interfaces, introduction of overloading and overriding methods.

We first provide an example of a complex test case that can not be easily generated by test generation tools. Then we detail our algorithm that adapts test cases for evolving software.

5.1 Motivating Example

When developers add a new class to extend the functionality of a software system, they need to also write new test cases. To write the test cases for the new class, often developers reuse test cases already available for similar elements, like sibling classes, that share some of functionality of the new class.

Classes belonging to the same hierarchy share common interfaces and behaviors, and differ only for some of the offered functionality. Software developers take advantage of these characteristics to develop test cases that often share the setup actions, for example the objects under tests are built by passing the same parameters, present same invocation sequences, and use oracles that inspect the same output values, but expect different results.

Figure 5.1 shows a test case for the class `CopticChronology` of the `JodaTime` library. In the version 1.2, revision 911, of the `Jodatime` library, developers added a class `EthiopicChronology` to the hierarchy of class `BasicFixedMonthChronology` as a sibling of class `CopticChronology`. TCA can reuse the test case for the class `CopticChronology` shown in Figure 5.1 to automatically generate the test case for the class `EthiopicChronology` shown in Figure 5.2. The test case iterates over all the days in the range 0 AC - 3000 AC, converts the day

```
48   private static final Chronology COPTIC_UTC =
         CopticChronology.getInstanceUTC();
305  DateTime epoch = new DateTime(1, 1, 1, 0, 0, 0, 0, COPTIC_UTC);
306  long millis = epoch.getMillis();
307  long end = new DateTime(3000, 1, 1, 0, 0, 0, 0, ISO_UTC).getMillis();
     [..]
311  DateTimeField monthOfYear = COPTIC_UTC.monthOfYear();
     [..]
     while (millis < end) {
       [..]
324    int monthValue = monthOfYear.get(millis);
       [..]
328    if (monthValue < 1 || monthValue > 13)
329      fail("Bad month: " + millis);
       // test era
       [..]
334    assertEquals("AM", era.getAsText(millis));
       [..]
       // test leap year
342    assertEquals(yearValue % 4 == 3, year.isLeap(millis));
```

Figure 5.1. A test case written by developers for class `CopticChronology`

representation from the ISO calendar to the Ethiopic calendar, then checks if the conversion is correct. This is a valid test case for the class `EthiopicChronology` that properly combines different objects and methods. TCA derives the value of the ethiopian month by retrieving the `DateTimeField` object that holds the month value (`monthOfYear` in line 307) from a `Chronology` object configured using an instance of class `EthiopicChronology` (`epoch` in line 301). The `DateTimeField` object gives the month in the Ethiopic Chronology that corresponds to a given timestamp (this conversion is done by invoking method `get`, line 320). Proper checks must be added, for example for the values of months and leap years (lines 324 and 342).

Notice that a test case with random invocations of methods of class `Ethiopic Chronology` would not be meaningful as the one generated by TCA. TCA can derive a lot of the domain information required to build good test cases from existing test cases, while competing techniques cannot.

```
 48 private static final Chronology ETHIOPIC_UTC =
      EthiopicChronology.getInstanceUTC();
301 DateTime epoch = new DateTime(1, 1, 1, 0, 0, 0, 0, ETHIOPIC_UTC);
302 long millis = epoch.getMillis();
303 long end = new DateTime(3000, 1, 1, 0, 0, 0, 0, ISO_UTC).getMillis();
    [..]
307 DateTimeField monthOfYear = ETHIOPIC_UTC.monthOfYear();
308 assertEquals(EthiopicChronology.EE, epoch.getEra());
    [..]
316 while (millis < end) {
        [..]
320     int monthValue = monthOfYear.get(millis);
        [..]
324     if (monthValue < 1 || monthValue > 13)
325         fail("Bad month: " + millis);
        [..]
330     assertEquals("EE", era.getAsText(millis));
        [..]
341     // test leap year
342     assertEquals(yearValue % 4 == 3, year.isLeap(millis));
```

Figure 5.2. A test case for class `EthiopicChronology` that TCA generated by adapting a test case of class `CopticChronology`.

5.2 Overview of Test Adaptation Process

The test adaptation process can generate test cases for a new class that extends a hierarchy of classes, for a new class that implements an interface, for a new method that overloads or overrides another method. In this section we use the term *element under test* to indicate the class or the method that has been recently added. To generate test cases for the element under test, TCA looks for code elements that are similar to the element under test, copies, and adapts the test cases of the similar elements to generate test cases for the new element. Figure 5.3 illustrates the three main steps of the adaptation process:

1. **Identify and Copy Candidate Test Cases:** TCA inspects the source code of the original software to identify code elements, either methods or classes, that are similar to the element under test. TCA then identifies the candidate test cases, i.e., the test cases that can be used to generate test cases for the element under test. The candidate test cases are all the test cases written to test the elements similar to the element under test.

2. **Adapt Candidate Test Cases**: Since candidate test cases are not written

for the new element, most of them are not directly applicable to test the new element, for example, they may suffer from compilation errors. In this step, TCA adapts the test cases to the element under test by:

(a) Updating references to the new element

(b) Resolving compilation errors

(c) Adapting the oracle of the test cases

(d) Repairing runtime failures

3. **Remove Redundant Test Cases**: TCA removes redundant and unusable test cases. TCA uses instruction coverage as a measure of removing redundant test cases.

TCA follows this process for all the *Test Reuse Patterns* presented in this chapter. In Sections 5.3 and 5.4, we detail the first two steps that are specific for each pattern, while in Section 5.5 we present step 3.

5.3 Identify and Copy Candidate Test Cases

When a new element is added to a software system, TCA looks for candidate test cases that can be adapted for the new element. TCA analyzes the source code to identify the elements related to the element under test, and identifies the test cases for the related elements. For example, if the new element is a new class added to a hierarchy of classes, TCA identifies the sibling and sub-sibling classes as the related elements, and considers the test cases written for these elements as candidate test cases. Below we describe the process to identify the related elements and select the candidate test cases for each *Test Reuse Pattern*:

5.3.1 Classes Added to a Hierarchy

When a new class is added to a hierarchy of classes, TCA automatically identifies the related elements as the classes in the same hierarchy of the new class, and selects all the test cases of these classes. The rationale behind this choice is that the classes in a hierarchy might share a common behavior, thus their test cases can be used to generate new test cases for the classes in the hierarchy. Figures 5.4 and 5.5 show the algorithms that identify and rank the best candidate classes.

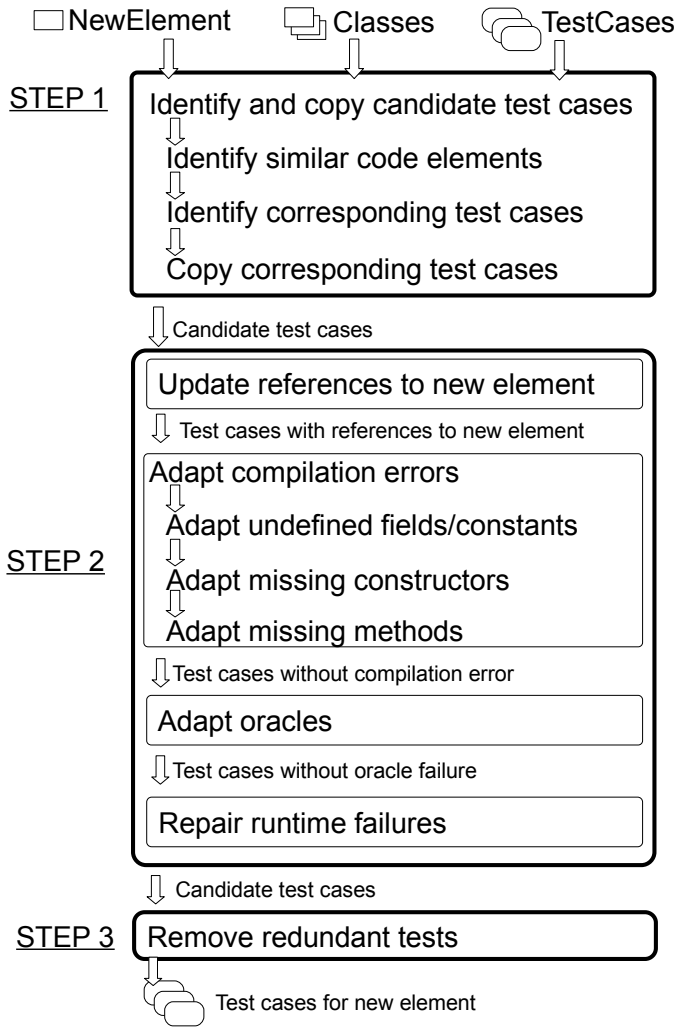

Figure 5.3. The process that generates test cases for new elements

The algorithm to generate candidate test cases shown in Figure 5.4 accepts the source code of the software system, S, the source code of the test cases, T, and the class under test, CUT, as input, and returns a list of candidate classes in the project workspace. The algorithm starts by identifying all the sibling and sub sibling classes of the class under test. If the *class* does not have any associated test case, TCA ignores that class. The method $identifyAllSiblingAndSubsibling Classes$ in Line 1 extracts the parent classes[1] of CUT and returns all the classes in the hierarchy of the parent class (except the CUT itself). Once TCA has identified the candidate classes, it computes a similarity score for all the classes

[1] TCA recursively traverses the hierarchy of CUT up to the `Object` class in Java

Require: S software code
Require: T test code
Require: CUT Class Under Test
Ensure: $newCandidateTestClasses$ a List of New Candidate Test Classes
1: $candidateClasses \leftarrow identifyAllSiblingAndSubsiblingClasses(CUT)$
2: $Map < class, score > classesWithScore$
3: **for all** $class$ in $candidateClasses$ **do**
4: **if** hasTestClass(class,T) **then**
5: $classesWithScore+ \leftarrow calculateSimilarityScore(class, CUT)$
6: **end if**
7: **end for**
8: $candidateClasses \leftarrow Sorter.sort(classesWithScore)$
9: **for all** $class$ in $candidateClasses$ **do**
10: $TestClass \leftarrow extractTestClass(class)$
11: $newTestClass \leftarrow createClassWithName(CUT.getName() + \text{“}Using\text{”} + TestClass.getName(), TestClass)$
12: $newCandidateTestClasses.add(newTestClass)$
13: **end for**
14: **return** newCandidateTestClasses

Figure 5.4. The algorithm that Identifies and selects the candidate test cases for a New Class

and ranks them based on their scores. The tests of the classes are prioritized according to the score of the related class (Lines 3-8). The algorithm to calculate the score (method $calculateSimilarityScore$ in Line 5) is shown in Figure 5.5. TCA stores the classes and their scores in variable $classesWithScore$ (Line 8). TCA then extracts the test cases of each candidate class using the method $extractTestClass$. This method identifies the class that contains the test cases for a given class passed as input. The method $extractTestClass$ identifies the test classes using a heuristic based on the common naming of Java test classes whose name contains the term "Test" either followed or proceeded by the name of the class under test. When looking for the test of a class A, $extractTestClass$ looks for a class named *TestA* or *ATest*. Next, TCA copies and renames the candidate test classes (lines 9-12). TCA renames each $candidateTestClass$ as follows:

<CUT>Using<TestClass>

Require: CUT Class Under Test
Require: $SimilarClass$
Ensure: an integer specifying similarity of $class$ and CUT
1: $NumberOfSameFields \leftarrow 0$
2: $NumberOfSameMethods \leftarrow 0$
3: **for all** $field$ in $class.getFields()$ **do**
4: **for all** $cutField$ in $CUT.getFields()$ **do**
5: **if** field.getName() == cutField.getName() **then**
6: $NumberOfSameFields++$
7: **end if**
8: **end for**
9: **end for**
10: **for all** $method$ in $class.getMethods()$ **do**
11: **for all** $cutMethod$ in $CUT.getMethods()$ **do**
12: **if** method.getName() == cutMethod.getName() **then**
13: $NumberOfSameMethods++$
14: **end if**
15: **end for**
16: **end for**
17: **return** $NumberOfSameFields+NumberOfSameMethods$

Figure 5.5. The algorithm that computes class similarity

For example, Figure 3.4 shows the hierarchy of the class $EthiopicChronology$. To generate test cases for the class $EthiopicChronology$, TCA selects the test cases of the classes `CopticChronology` and `IslamicChronology` as candidate test cases (the classes `BasicFixedMonthChronology` and `BasicChronology` are abstract and do not have any test case). TCA copies and renames the candidate test cases to prevent classpath conflicts. For instance, TCA generates a test class with the name `EthiopicChronologyUsingCopticChronologyTest` that is an exact copy of `CopticChronologyTest`.

The similarity of classes is calculated using the method $calculateSimilarityScore$ that is shown in Figure 5.5. The algorithm requires two classes as input and returns an integer value which quantifies the similarity of $class$ and CUT. The algorithm calculates the similarities between two classes as the number of fields and methods with the same name. The steps in lines 3-9 calculate the similarity score for the fields of the two classes, and store them in the variable $NumberOfSameFields$. The steps in lines 10-16 calculate the similarity score for the methods of the classes, and store them in the variable

$NumberOfSameMethods$. Finally, the algorithm returns the similarity score.

5.3.2 Interface Implementations

To generate test cases for a new class that implements an interface, TCA follows same algorithms depicted in Figures 5.4 and 5.5 with the difference that in Line 1 of Figure 5.4, TCA calls the method $identifyInterfaceImplementations$ instead of $identifyAllSibilingAndSubSiblings$. The method $identifyInterface Implementations$ identifies the classes that implement the same interface of the new class and consider them as candidate classes. To this end, TCA extracts the fully qualified names of the interfaces declared by new element, and traverses all the classes to find classes implementing the same interfaces. Then, for each of the similar elements, TCA identifies their test cases and copy them as test cases of the new element. TCA renames each test class as follows:

<ClassUnderTest>Using<TestClassToAdapt>

where *<ClassUnderTest>* is the name of the class to test, and *<TestClassToAdapt>* is the name of the test class that is copied.

5.3.3 Overloaded Methods

An overloaded method is a method that shares the name with other methods but has different parameters. TCA generates test cases for a new overloaded method following the algorithm shown in Figure 5.6. The algorithm requires the source code of the software system, S, the source code of the test cases, T, and the method under test, MUT as inputs, and returns a list of candidate test classes. Candidate test classes contain the test cases that can be used to test the new overloaded method.

The algorithm starts by identifying the overloaded methods, $overloaded Methods$, by calling method $identifyOverloadedMethods$ in Line 1. The method $identifyOverloadedMethods$ invokes the function $identifyAllSibl ingAndSubsibling$ to identify the classes in the same hierarchy of the modified class, then identifies all the methods with the same name of the MUT and appends them to the list of overloaded methods.

For each overloaded method, TCA identifies all the test classes that test the overloaded method by calling $extractTestClasses$ (Line 4). The method $extract TestClasses$ returns all the test cases that invoke MUT in the test suite of the project. The method $extractTestClasses$ receives as input a method m and returns a set of test cases that cover this method. The method $extractTestClasses$

Require: S software code
Require: T test code
Require: MUT New Overloaded Method
Ensure: $newCandidateTestClasses$ a List of New Candidate Test Classes
1: $overloadedMethods \leftarrow identifyOverloadedMethods(MUT)$
2: **for all** $method$ in $overloadedMethods$ **do**
3: **if** $hasTestMethod(method)$ **then**
4: $testClasses \leftarrow extractTestClasses(method)$
5: **for all** $testClass$ in $testClasses$ **do**
6: $newTestClass \leftarrow createTestClass(\text{“}testOverload\_\text{”} + MUT.getClass().getName() + MUT.getSignature() + \text{“}\_Using\_\text{”} + testClass.getName() + \text{“}\_\text{”} + method.getSignature(), testClass)$
7: $newCandidateTestClasses.add(newTestClass)$
8: **end for**
9: **end if**
10: **end for**
11: **return** $newCandidateTestClasses$

Figure 5.6. The algorithm that Identifies and selects candidate test cases for Overloaded Methods

simply returns all the test cases that contain at least one invocation of the method m.

Next, the algorithm iterates over all the test classes and generates new test classes by calling the method $createTestClass$ (Line 6). For all the candidate test methods that belong to the same test class, TCA copies the test methods and the accompanying `setUp` and `tearDown` methods as a test class for the new overloaded method. TCA names the test classes as follows:

$$testOverload\_ < ClassUnderTest > \_ < Signatureof MUT >$$

$$\_Using\_ < UsedTestClass > \_ < SignatureofOverloadedMethod >$$

Where $ClassUnderTest$ is the class of new method, $Signatureof MUT$ is the signature of the new method, $UsedTestClass$ is the test class that contains the tests for testing the overloaded method, $SignatureofOverloadedMethod$ is the signature of the overloaded method. TCA stores the references to the newly generated test classes in $newCandidateTestClasses$.

5.3.4 Overridden Methods

An overridden method is a new method with the same signature of a method defined in a parent class. TCA generates test cases for a new overridden method by reusing the test cases of the base methods[2] from parent class. More specifically, TCA extracts the signature of the new overridden method and looks for methods with the same signature in the parent class. If TCA finds some base methods in the parent class, it looks for the test cases for the base methods, and consider them as candidate test cases. The algorithm is shown in Figure 5.7.

Require: S software code
Require: T test code
Require: MUT New Overridden Method
Ensure: $newCandidateTestClasses$ a List of New Candidate Test Classes
1: $overriddenMethods \leftarrow identifyOverriddenMethods(MUT)$
2: **for all** $method$ in $overriddenMethods$ **do**
3: **if** hasTestMethod(method) **then**
4: $testClasses \leftarrow extractTestClasses(method)$
5: **for all** $testClass$ in $testClasses$ **do**
6: $newTestClass \leftarrow createTestClass(\text{“}testOverride\_\text{”} + MUT.getClass().getName() + method.getSignature() + \text{“}\_Using\_\text{”} + +method.getSignature(), testClass)$
7: $newCandidateTestClasses.add(newTestClass)$
8: **end for**
9: **end if**
10: **end for**
11: **return** $newCandidateTestClasses$

Figure 5.7. The algorithm that identifies and selects the candidate test cases for overridden methods

The algorithm first identifies overridden methods, $overriddenMethods$, by calling method $identifyOverriddenMethods$ in Line 1. For each overridden method, TCA calls $extractTestClasses$ to extract the test classes associated with the method (Line 4). Method $extractTestClasses$ returns all the test cases that invoke MUT in the project's test suite. To extract the test cases, the algorithm extracts test cases that invoke MUT at least once. Next, the algorithm iterates over all test classes and generates new test classes by calling method $createTestClass$ as it is shown in Line 6. For all candidate test methods that

[2]A base method is a method with the same signature of the new method in parent class

belong to the same test class, TCA copies test methods and accompanying `setUp` and `tearDown` methods as a test class for new overridden method. TCA creates candidate test cases to avoid conflict in the project using the following pattern:

$$testOverride\_ < ClassUnderTest > \_ < Signatureof MUT >$$

$$\_Using\_ < UsedTestClass > \_ < SignatureofOverrriddenMethod >$$

$ClassUnderTest$ is the class of MUT, $Signatureof MUT$ is the signature of the new method, $UsedTestClass$ is the test class that contains the test for the overridden method, $SignatureofOverriddenMethod$ is the signature of the overridden method. Finally, TCA stores the references to newly generated test classes in $newCandidateTestClasses$.

5.4 Adapt Candidate Test Cases

With this step, TCA adapts the candidate test cases identified in the previous step to test the new elements. For each candidate test case, TCA first updates all the references to the element under test, then solves compatibility issues raised by compilation errors, adapts the oracle of the test cases, and finally repairs runtime failures raised while running the test cases.

5.4.1 Update References to New Element

TCA looks for all the references to the old element in the candidate test cases and replaces them with the new element. Moreover, TCA imports the package definitions required by each candidate test class.

For example, when applied to the test case for class `EthiopicChronology` shown in Figure 5.2, TCA updates all the occurrences of the term `CopticChronology` with the term `EthiopicChronology`. To improve the readability of the new test cases, TCA updates also all the variable names that contain portions of the names of the original class under test with portions of the name of the new class under test (we split names according to the Java camel case convention). In the example of Figure 5.2 TCA replaces `COPTIC_UTC` in lines 48 and 307 with `ETHIOPIC_UTC`. TCA identifies the class under test using a simple heuristic based on the standard naming of JUnit test cases for Java, i.e., it assumes that the name of a test class is prefixed or followed by the keyword “Test”, and thus identifies

the class under test by removing the prefix/suffix “Test” from the name of the candidate test class.

5.4.2 Adapt Compilation Errors

The generated test cases may lead to compilation errors because of undefined fields, constants, constructors, or methods. TCA fixes these compilation errors by applying the algorithms described in this section. According to our experience, other types of compilation errors are not frequent in candidate test cases, thus, TCA comments all other types of compilation errors. In the following subsections, we describe how TCA addresses each incompatibility.

Adapt Undefined Fields and Constants

Test cases often contain references to constants or fields declared in the classes under tests. TCA updates references to the constants and fields of the original class under test by replacing them with references to the corresponding constants and fields declared in the class under test. TCA identifies the corresponding fields and constants by applying the algorithm $corresponding(S_x, S_y, occ_{ax})$ to find the corresponding terms as discussed in Section 4.4. For example, to adapt the test case of Figure 5.2, TCA replaces the references to `CopticChronology.AM` with references to `EthiopicChronology.EE` (see line 308), the corresponding constants used to indicate the default era. Both calendars present a constant that indicates the default era, but the name of this constant is different, because each calendar uses a specific term to indicate era, AM stands for “Anno Martyrum” while EE stands for “Ethiopian Era”.

Adapt Undefined Constructors

Candidate test cases may suffer from compilation errors due to undefined constructors. TCA finds corresponding constructors in the new element and adapts the parameters for the new constructor.

Figure 5.9 shows the algorithm $FindSimilarConstructor$ that accepts two arguments, the incompatible constructors of the similar class, $consSimilarClass$, and the class under test, CUT, and returns the signature of the most similar constructor belonging to CUT. First, the algorithm extracts the compatible constructors of CUT in Line 2 by calling the method $findCompatibleConstructor sofClass$. Since constructors that are present in the subclasses of CUT are also compatible with the class instantiation, the method $findCompatibleConstruct$

orsofClass finds the constructors that are present both in *CUT* and in its subclasses by exploring the project under test. Once TCA identified the candidate constructors, TCA calculates the similarity of each constructor with *consSimilarClass*. Figure 5.10 details the method *calculateSimilarityScore* in Line 4. Next, the algorithm sorts the candidate constructors based on their similarity scores (Line 6) and returns the most similar constructor.

The method *calculateSimilarityScore* (shown in Figure 5.10) receives two inputs: the constructors of the new class, *constructorNewClass*, and the constructor of the similar class, *constructorSimilarClass*. *calculateSimilarityScore* returns an integer value that quantifies the similarity of the two constructors (Lines 4-6 and 7-9). The method *isCompatible* in Line 7 returns a boolean value that indicates the compatibility of the parameter types of the two constructors. The compatibility condition indicates if the parameter of the similar constructor can be replaced with the parameter of the new constructor. The algorithm extracts all the subtypes and the implementing types of the parameter in the new constructor, and considers the parameter of the similar constructor as compatible only if the type of this parameter belongs also to the extracted subtypes.

Figure 5.8 shows the excerpt of a test case for the class `SVGOutput` of Barbecue v1.5 that TCA generated by adapting the test cases for the class `GraphicsOutput`. The constructor of the class `GraphicsOutput` used in the candidate test case receives four objects of the following types `Graphics2D`, `Font`, `Color` and `Color`. Class `SVGOutput` does not provide a constructor that receives the same input types, thus the candidate test case causes a compilation error.

a)
```
g = new GraphicsMock();
output = new SVGOutput(
    g,
    DefaultEnvironment.DEFAULT_FONT,
    fgColour, bgColour);
```

b)
```
output = new SVGOutput(
    new JavadocEscapeWriter(EasyMock.createMock(Writer)),
    DefaultEnvironment.DEFAULT_FONT,
    fgColour, bgColour,
    0, "" );
```

Figure 5.8. An example of constructor call adaptation.

To repair these errors, TCA replaces the original constructor calls with the constructor of the class under test that is the most similar to the replaced con-

structor. TCA selects the constructor with the highest rank according to the method *calculateSimilarityScore* described in Figure 5.10.

Require: $consSimilarClass$ incompatible constructor of similar class
Require: CUT Class Under Test
Ensure: most similar constructor in Class Under Test
1: $Map < constructor, score > consScore$
2: $constructors = findCompatibleConstructorsofClass(CUT)$
3: **for all** $cons$ in $constructors$ **do**
4: $consScore+ \leftarrow calculateSimilarityScore(consSimilarClass, cons)$
5: **end for**
6: $candidateConstructors \leftarrow Sorter.sort(consScore)$
7: **return** $candidateConstructors[0]$

Figure 5.9. Algorithm to Find Similar Constructor

Require: $constructorSimilarClass$ constructor of similar class
Require: $constructorNewClass$ constructor of Class Under Test
Ensure: $score$ similarity of two constructors
1: $socre \leftarrow 0$
2: **for all** $parSimilar$ in $constructorSimilarClass.getParameters()$ **do**
3: **for all** $parNew$ in $constructorNewClass.getParameters()$ **do**
4: **if** $parSimilar.getName() == parNew.getName()$ **then**
5: $Score++$
6: **end if**
7: **if** $isCompatible(parSimilar.getType(), parNew.getType())$ **then**
8: $Score++$
9: **end if**
10: **end for**
11: **end for**
12: **return** score

Figure 5.10. Algorithm to Calculate Similarity of Constructors

After finding a similar constructor, TCA looks for parameters shared by the constructor used in the original test and the similar constructor. The matching is done by iteratively looking for parameters of the two constructors with the same type or name, with priority to the types. TCA reuses the compatible parameters of the original test case by positioning them properly in the new constructor call.

In the case of mismatching parameters, TCA generates input values as follow. If the parameter is primitive TCA uses some default values: "0" for numeric types and bytes, the space character for the `char` type, the empty string for type String, and an array with a default element for type array. If the required parameter is an object, TCA uses either some constants of that type, or some factory methods that return an object of that type. If none is found, TCA invokes a constructor of the required type[3]. If the constructor requires object parameters, TCA creates suitable stubs. Our Java implementation uses Easymock[4]. Figure 5.8.b shows how TCA creates the first parameter of the `SVGOutput` constructor, which is of type `Writer`: TCA invokes the constructor of the class `JavadocEscapeWriter`[5], with a stub created using Easymock.

Adapt Undefined Method

Sometimes method invocations cause compilation errors in candidate test cases. When TCA identifies a call to a non-existing method, it looks for a similar method in the class under test. It finds the most similar method by considering the signatures of all the methods declared in the class under test. Once TCA identifies the most similar method, TCA replaces the invocation of the original method with the new one following the same steps adopted in the case of constructors: It adds reusable parameters, and then looks for new parameters.

Figure 5.11 shows the algorithm $findSimilarMethod$ that finds the most suitable method when either a *parameter mismatch* or an *undefined method* compilation error occurs. The algorithm requires two inputs: the method signatures of the similar class and of the class under test. The algorithm $findSimilarMethod$ returns the most similar method in class under test. First, the algorithm finds the public and protected methods of the class under test in Line 2, by calling method $findPublicMethodsofClass$. The algorithm selects all the methods with public and protected declaration, and returns them as result. Then, the algorithm iterates over extracted methods and calculates the similarity score of each method by calling $calculateSimilarityScore$ (described in Figure 5.10) and stores them in a Map data structure called $methScore$ (Lines 3-5). Next the algorithm sorts the methods based on their scores (in Line 6).

In some cases, a class may have a method with the same goal but a different name. To handle this situation the similarity score is calculated on the basis

[3]In the case of interfaces, it randomly picks up a constructor of the class that implements the interface

[4]http://www.easymock.org/

[5]Class `JavadocEscapeWriter` implements interface `Writer`

of both the method name and the parameter types. TCA considers the fact that methods belong to the same class may present different names but same goal, for example method $initialize(int)$ may have the same meaning of method $init(int)$. The algorithm computes the similarity score by calculating the average Levenshtein distance of the candidate methods with $methSimilarClass$. The method $getAverageLevenshteinDistance$ calculates the average Levenshtein distance, $aveLD$, based on the classic Levenshtein distance algorithm [Lev65]. TCA starts from the most similar methods of the class under test and returns the first method that has a smaller Levenshtein distance than $aveLD$ (Lines 7-12). In this way, TCA considers also the name of the methods to extract the most similar one. For example, in Figure 5.12 the method `yearsBetween` raises an undefined method compilation error since it is copied from the test suite of the class `Seconds`. This test case has been generated by adapting the test `testFactory_yearsBetween_RInstant` defined in class `TestYears`, which tests class `Years`. The method that calculates the difference in terms of years between two dates, `yearsBetween(DateTime,DateTime)`, declared in class `Year`, is no longer present in class `Seconds`, the class under test. Class `Seconds` instead implements the method `secondsBetween(DateTime,DateTime)` that calculates the seconds between two dates. Since the goal of the test is to evaluate the proper implementation of the diff functionality, the method `secondsBetween` can be considered a suitable replacement candidate for the method `yearsBetween` in the test for class `Seconds`. The Levenshtein distance between the names of these two classes are less than the one of other methods, and so TCA selects this method as the most similar one. If no method is found in Lines 7-12, TCA returns the first item in the list of sorted candidate methods.

5.4.3 Adapt Oracles

Even if the candidate test cases are free from compilation errors at this stage, some of them may present runtime errors. The generated test cases could be affected by two kinds of execution problems: assertion failures and runtime exceptions.

In case of failing assertions, TCA repairs the failure by adapting the assertion to expect the actual behavior of the test case. To this end, TCA uses a state-of-the-art tool called ReAssert that repairs oracles by modifying expected part of the assertions.

To repair a single failure in a test, ReAssert first instruments the test classes to record the values of the method arguments for the failing assertions. It then re-executes the test and catches the failure exception that contains both the stack

Require: $methSimilarClass$ incompatible method of similar class
Require: CUT Class Under Test
Ensure: most similar method in Class Under Test
1: $Map < method, score > methScore$
2: $methods = findPublicMethodsofClass(CUT)$
3: **for all** $meth$ in $methods$ **do**
4: $methScore.put \leftarrow calculateSimilarityScore(methSimilarClass, meth)$
5: **end for**
6: $candidates \leftarrow Sorter.sortMethod(methScore)$
7: $aveLD = getAverageLevenshteinDistance(candidates, methSimilarClass)$

8: **for all** $meth$ in $candidates$ **do**
9: **if** $levenshteinDistance(meth.getName(), methSimilarClass) < aveLD$ **then**
10: $return\ meth$
11: **end if**
12: **end for**
13: **return** $candidates[0]$

Figure 5.11. Algorithm to Find Similar Method

trace and the recorded values. It next traverses the stack trace to find the code to repair and examines the structure of the code and the recorded values to change the code properly. It finally recompiles the code changes and repeats these steps if the test has another failure.

5.4.4 Repair Runtime Failures

Test cases that throw an exception are a special case of failing test cases. Exceptions may indicate an error in the implementation of the class, wrong test inputs, or an invalid test case setup. Existing techniques like Randoop [PLEB07] exclude test cases raising exception, however these test cases might exercise faulty behaviors that should thus be inspected by software developers. When the test case execution raises an exception, we rely on inspection by the developers to determine if the test cases should be removed or kept because they may pinpoint a fault.

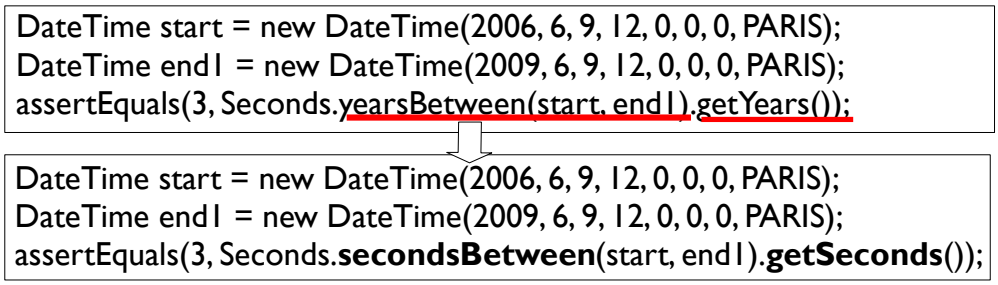

Figure 5.12. An example of method invocations repair. TCA repairs the compilation error caused by the invocation of method `yearsBetween` by invoking the corresponding method, `secondsBetween`.

5.5 Removing Redundant Test Cases

TCA generates test cases reusing test cases written for different classes, thus multiple test cases might cover the same software behavior. To prune duplicate test cases, TCA adopts a simple heuristic that consists of executing the test cases and measuring the instructions covered during execution[6]. TCA discards the test cases that do not increase the instruction coverage, i.e., that do not cover instructions not already covered.

TCA picks all the test cases generated for a specific element. Then, TCA executes each test case and records the instructions covered by test case on the element under test. When TCA executes another test case, TCA compares the instruction covered by new test case with recorded instructions already covered. If new test case is covering new instructions, TCA keeps new test case otherwise ignores it. TCA performs this process for all the generated test cases and shows the final results to the developers.

[6]The current prototype uses EclEmma, www.eclemma.org

Chapter 6

Prototype Implementation

To evaluate the effectiveness of our approach we developed a prototype implementation called also TestCareAssistant (TCA). The prototype is implemented as an Eclipse plug-in that applies our technique to both repair a compilation error in a test case and generate a test case for a new class or method. This chapter provides some details of the design and the implementation of this eclipse plug-in.

TCA extends the eclipse interface by adding a contextual menu to the project explorer in Eclipse IDE, as shown in the screenshots presented in Figure 6.1.

In the next two sections, we presents the implementation details of the "change of method declaration" *Test Reuse Pattern* and the other *Test Reuse Patterns*.

6.1 The Test Repair Toolsuite

The test repair tool suite implements the "change of method declaration" *Test Reuse Pattern*. The toolsuite takes as input the source code of the original and the modified software and a set of test cases for the original software that suffer from compilation errors caused by the changes, and repairs the broken test cases for the modified software.

TCA works in two different scenarios:

1. The developer selects TCA by invoking the regular Eclipse quick assistant function (*ctrl+1*) to see the suggestion proposed by TCA and apply TCA to the broken test case. Figure 6.2 shows a suggestion proposed by TCA for repairing a compilation error of a test case.

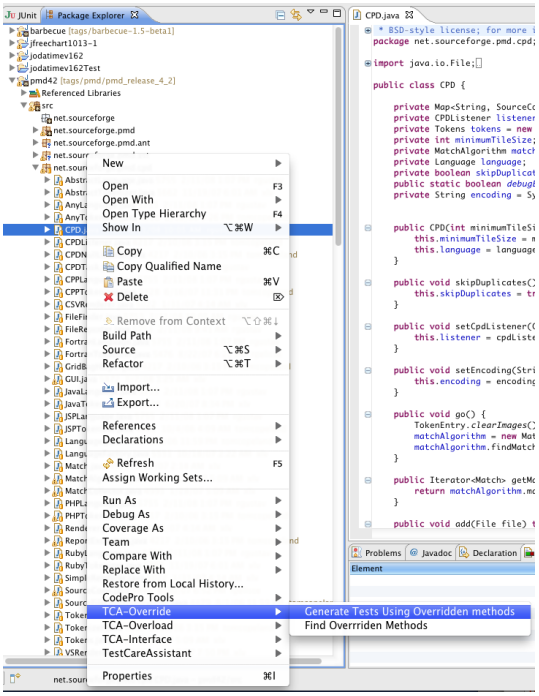

Figure 6.1. A screenshopt of the TCA Eclipse plugin

2. The developer selects the TCA repair context menu by right clicking on a set of test cases affected by compilation errors (can be a test class, a package, or a folder containing packages). TCA then examines each single compilation error, and, if it finds any repair, it corrects the broken test case. Then the developer can inspect the repairs proposed by TCA.

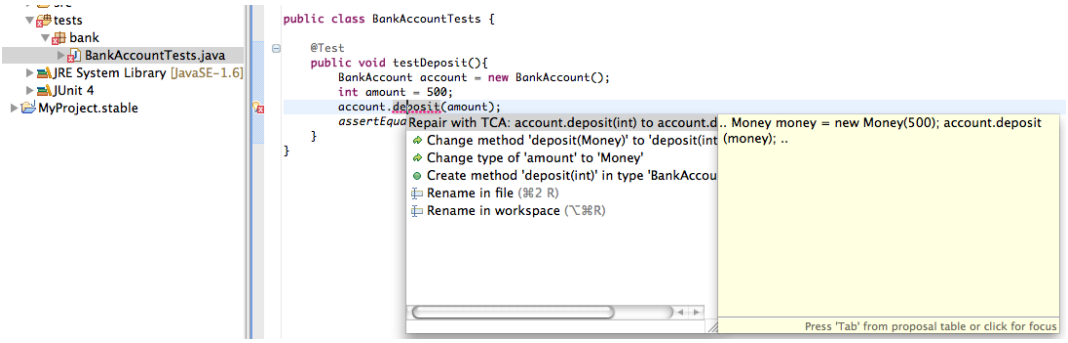

Figure 6.2. TCA Test Repair Screenshot

Figure 6.3 shows an overview of the architecture of the test repair tool suite. The figure shows the components, the external libraries and their dependencies.

The *Eclipse UI* is a component of Eclipse that handles the developer interaction. In fact, the *Eclipse UI* component delivers the change information that includes the source code of the original and the modified software and the obsolete test case to the *Change Analyzer* component.

The *Change analyzer* component detects the type of change in the method signature. *Change analyzer* uses *JDiff*, as explained in details later in this Chapter, to extract the types of change that are parameter type change, return type change, parameter addition, and parameter removal. *Corresponding Variable Finder* finds the initialization variables that are required to repair the test case using *Soot* and *DaTeC* (these two components are described later in this Chapter).

Dynamic Inspector finds the values for initializing the variables using *Soot Instrumentor* and *Junit Runner*. *Soot Instrumentor* augments the byte code of the test case to record the values of specific variables at a given location. *Junit Runner* runs the instrumented test case and dumps the values of the variables in an external file. *Test Repair Generator* produces the suggested test repair that will be inserted into the test code.

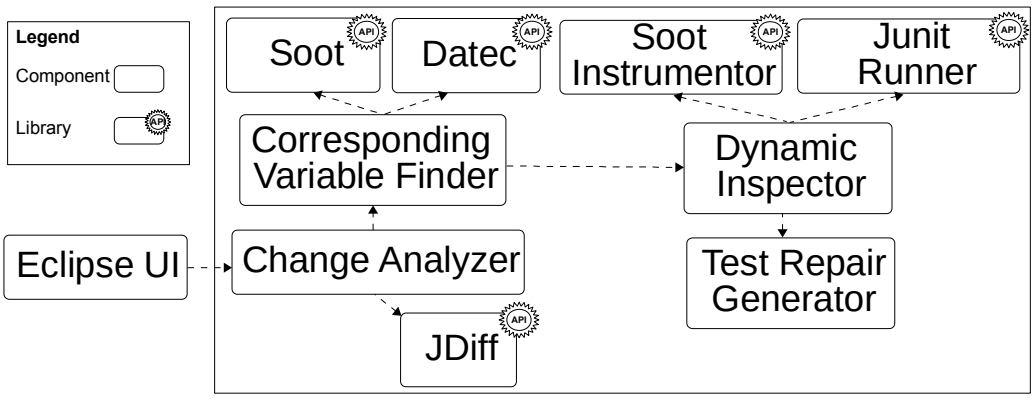

Figure 6.3. The architecture of TCA Test Repair

The prototype tool uses three third party open-source components to perform background operations: JDiff[1], a Java source code differencing tool, *DaTeC* [GAM09], an interprocedural data flow analysis tool, and *Soot* [VRCG+99, GH01], a static data flow analyzer:

JDiff: identifies the differences between two APIs, and indicates all the packages, classes, constructors, methods, and fields that have been removed,

[1]http://www.jdiff.org/

added, or changed. TCA uses the changes related to method and field declarations. The changes identified with JDiff allow TCA to detect the type of changes. JDiff is unable to find the corresponding line in original version of the software, thus TCA uses the Needleman-Wunsch algorithm [NW70], which performs a global alignment on the body of the original and the modified methods to identify the corresponding lines and the modified variables. TCA uses the information extracted with JDiff and with the Needleman-Wunsch techniques to repair the broken test cases.

***DaTeC*:** computes contextual data flow coverage of Java programs by executing the Java bytecode suitably instrumented to record the coverage of contextual def-use associations. *DaTeC* identifies the data about contextual data flow at different levels of granularity, from the finest grain level that presents all covered and not-yet-covered associations, to the coarsest granularity level that summarizes the amount of covered pairs for selected classes only.

Soot: is a framework for optimizing Java bytecode. The framework is implemented in Java and provides a set of intermediate representations and a set of Java APIs for optimizing Java bytecode directly. The optimized bytecode can be executed using any standard Java Virtual Machine (JVM). We used Shimple, which is one of the intermediate representations of Java bytecode to extract information about Java classes [VRHS+99].

6.2 The Test Evolution Toolsuite

This section provides some implementation details of *Test Reuse Patterns*: Extension of class hierarchy, Implementation of Interface, Introduction of overloaded and overridden methods. TCA enriches the context menu of Eclipse with a menu item for each *Test Reuse Pattern*. When a developer selects one of these menu items, TCA generates test cases for the class or the method under test and shows the results.

Figure 6.4 shows a high-level architecture of our prototype test evolution toolsuite. The *Eclipse UI* component receives the developer actions and passes them to TCA that generates the test cases for the selected items. *Element Extractor* uses JDT[2], described later in this chapter, to obtain selected elements by the developer.

[2]Eclipse Java Development Tools (JDT): www.eclipse.org/jdt/

The *Similarity Detector* component is responsible for finding the elements similar to the selected ones. TCA uses sibling and sub sibling classes for new class elements and overloading/overriding methods for new method elements. For each similar element, *Test Finder* seeks for test cases written for similar elements. *Test Finder* uses different mechanisms for classes and methods. The *Test Finder* component identifies test cases for classes by searching for entities with the same name as the class name augmented with the "Test" prefix or postfix. *Test Finder* extracts test cases for new methods by finding all the test cases that call the current method.

The *Test Adaptor* component is the core component of TCA. It copies candidate test cases, resolves compilation errors (using *JDiff* and *Compiler* libraries), fixes assertion failures (using *ReAssert*), removes runtime errors (using *Junit Runner*), and optimizes the number of test cases (using *Eclemma Coverage* component). The generated test cases will be shown to the developer for further modification if it is required.

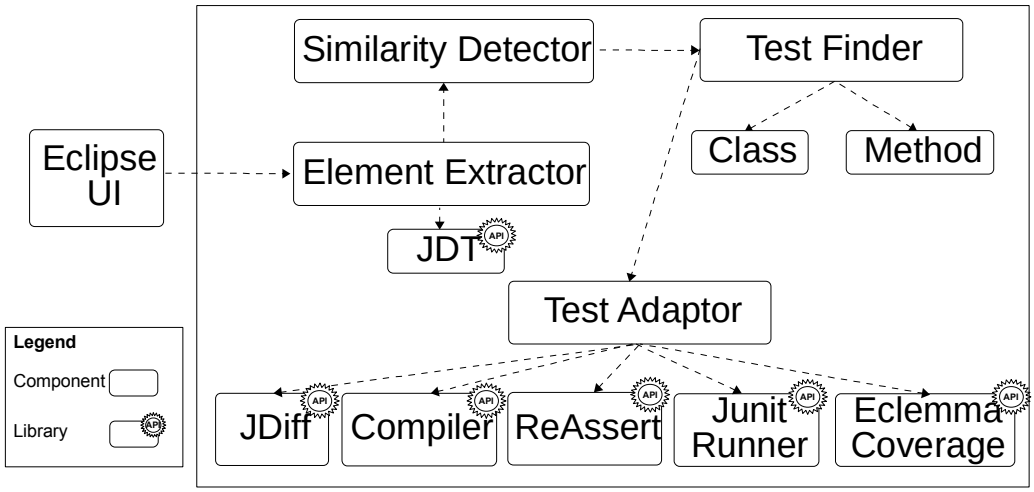

Figure 6.4. The architecture of the TCA Test Adaptation Toolsuite

TCA uses Eclipse JDT as a library to extract structure of the project source code. The Eclipse Java Development Tools (JDT) provides APIs to access and manipulate Java source code. JDT allows to access the existing projects in the workspace, create new projects and modify and read existing projects. It also allows developers to launch Java programs. Java JDT allows to access Java source code via two different means: The Java Model and the Abstract Syntax Tree (AST).

Java Model: A Java Model provides the API for navigating the Java element

tree. The Java element tree defines a Java centric view of a project. It provides elements like package fragments, compilation units, binary classes, types, methods, fields. Each Java project is represented in Eclipse as a Java model. The Eclipse Java model is a light-weight and fault-tolerant representation of the Java project. It does not contain as much information as the AST (for example it does not contain the body of a method) but it can be quickly recreated in the case of changes. For example the outline view in Eclipse uses the Java model for its representation. In this way the information in the outline view can be quickly updated. The Java model is defined in the package and plugin "org.eclipse.jdt.core".

Abstract Syntax Tree (AST): The AST is a detailed tree representation of the Java source code. The AST defines an API to modify, create, read, and delete source code. The package for AST is org.eclipse.jdt.core.dom in the Eclipse org.eclipse.jdt.core plugin. Each element in the Java source file is represented as a subclass of ASTNode. Each AST node provides specific information about the object it represents. For example, MethodDeclaration represents methods, VariableDeclarationFragment represents variable declarations, and SimpleName represents any string which is not a Java keyword. The AST is created based on an ICompilationUnit from the Java Model.

Since TCA works at different level of granularity depending on each pattern, TCA uses both the Java model and the AST components to extract class level, method level, and statement level information about the source code.

Here we describe a scenario of class level source data extraction. TCA needs to extract sibling and sub sibling classes for the "extension of class hierarchy" *Test Reuse Pattern*. Since TCA only needs information at the granularity level of class, TCA uses the Java model to find sibling classes of a given class. First of all TCA finds the superclass of the current class, then TCA extracts all classes extending the super class. This gives TCA a list of ITypes that represent sibling classes of the given class.

On the other hand, to create and edit the source code of test cases, TCA needs the AST of the component to extract appropriate information. For example, to extract all method calls in a given test method, TCA implements a visitor class that extracts all method invocation objects in an AST representation of a given test method. In this case TCA is only interested in method calls, so TCA ignores all other information available on AST.

Chapter 7

Evaluation

This chapter describes the methodology adopted to validate the proposed approach for test suite evolution, *TestCareAssistant*, and presents the evaluation results obtained by applying the *Test Reuse Patterns* on some case studies.

The results obtained in the experiments indicate that *TestCareAssistant* is a viable solution to evolve test suites.

7.1 Evaluation Methodology

The goal of the empirical evaluation is to verify both the applicability and the effectiveness of TCA in evolving test suites by addressing the following questions:

RQ1 How frequently can *Test Reuse Patterns* be applied to evolve test cases?

RQ2 How effective are the test cases adapted by TCA? How do these test cases compare with the test cases generated by state-of-the-art techniques?

RQ1 refers to the applicability of *Test Reuse Patterns*. To address this question, we apply TCA to a set of software systems developed by different organizations, and measure the frequency of software elements (classes or methods) whose test cases could be automatically repaired or generated with the provided *Test Reuse Patterns*. We evaluated the results by categorizing them on the basis of the *Test Reuse Patterns*.

RQ2 evaluates the effectiveness of TCA by focusing of the capability of the test cases generated by TCA in covering software behaviors. To address this research question, we compared the test cases generated by TCA with the ones generated by both software developers and some state-of-the-art test case generation tools. In the case of "*change of method declaration*", we compared the results of TCA

only with the test cases produced by developers, because, to the best of our knowledge, there is no tool (other than TCA) for repairing test cases broken by changes in method declarations.

We compare the test cases generated with TCA with test cases produced by developers manually for the pattern "*change of method declaration*": the comparison is easy in this case, because TCA simply repairs the way variables are initialized thus leading to test cases that shall be the same as the ones repaired by developers, or very similar to them. We discuss the measures of similarity in Section 7.4. For the other *Test Reuse Patterns* we address *RQ2* using two measures: Code Coverage and Conciseness:

Code Coverage Structural code coverage is a measure commonly adopted to evaluate the effectiveness of test generation tools. High code coverage increases the confidence about the quality of the code under test. There are many code coverage criteria proposed in the literature such as statement, branch, condition, condition/decision, etc. In our comparison we adopted the statement coverage criterion that measures the percentage of statements executed by at least one test case.

We compared the code coverage obtained with the test cases generated with TCA with the code coverage of test cases generated by project developers, and by three well known test case generation techniques:

- Randoop version 1.3.2: http://code.google.com/p/randoop
- Google CodePro version 7.1.0: https://developers.google.com/java-dev-tools/codepro/doc/
- EvoSuite version 20110929: http://www.st.cs.uni-saarland.de/evosuite/

Conciseness Many test case generation tools aim to achieve a high level of coverage for code under test. However, the fact that these test cases should be readable is often neglected in the state-of-the-art techniques. We claim that our approach can generate more concise test cases by optimizing number of test cases, thus making the final test cases easier to read and inspect. In fact, the third step of TCA removes redundant test cases and ignores test cases that do not contribute to the increase of the code coverage. We compare TCA with state-of-the-art techniques by comparing number of test methods generated as the test suite of each element under test.

The remainder of this chapter presents our case studies and the empirical results obtained by applying each *Test Reuse Pattern* to the case studies.

Subject	LOC	Download	Classes	Test Classes	Test Methods
Xstream 1.31	24,655	N/A	218	455	1486
PMD 4.2	65,279	747,765	483	193	880
Barbecue 1.5b1	8,842	126,736	55	20	294
JodaTime 1.62	63,922	207,323	99	204	4,402
JFreeChart 1.013	217,357	48,235	471	410	2,686

Table 7.1. Subject Programs

7.2 Case Study Subjects

We chose five open-source projects to evaluate applicability and effectiveness of TCA. Table 7.1 presents some details about the popularity and complexity of the case studies by giving: the number of lines of source code (column *LOC*), the number of downloads[1] (column *Download*), the number of classes (column *Classes*), the number of test classes (column *Test Classes*) and the number of test methods (column *Test Methods*). The numbers are calculated with the Metrics 1.3.6 Eclipse Plugin[2].

We selected these software projects because they are good representatives of software projects on which TCA could be applied: they include 3 libraries and 2 GUI applications. All the projects come with manually written test cases that cover the software behavior and are kept up to date by software developers.

JFreeChart

JFreeChart[3] is an open-source framework that creates a wide variety of interactive and non-interactive charts. JFreeChart can be used to create XY, Pie, Gantt, and Bar charts and many other specific charts such as wind, polar, and bubbles charts.

JFreeChart supports many output types including swing components, image files, and vector graphics file format. JFreeChart has been an active project since February 2000, and all the revisions are accessible from the SVN repository on SourceForge[4].

[1] the data have been collected in August 2012
[2] http://metrics.sourceforge.net/
[3] http://www.jfreechart.org
[4] http://www.sourceforge.net/

JodaTime

JodaTime[5] is a replacement for Java date and time classes. JodaTime provide utility classes to manage different types of multiple calendar systems: Gregorian, Julian, Buddhist, Coptic, Ethiopic, and Islamic. This project is used in many research papers as case study [BZ11, JOX10b, JOX10a, FZ12, DR12].

Barbecue

Barbecue[6] is a Java library that supports the creation, printing, and displaying of barcodes in Java applications. Barbecue supports many barcode formats as well as a number of predefined formats. Barcodes can be exported to image and SVG formats. The project is the smallest project that we considered and consists of about 9 KLOC.

PMD

PMD[7] is a source code analyzer that scans the Java source code to find potential problems like possible bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code. PMD has been integrated in many IDEs such as JDeveloper[8], Eclipse[9], Jedit[10], IntelliJ IDEA[11], Ant[12], and Emacs[13].

Xstream

Xstream[14] is a library to serialize objects to XML and deserialize XML into objects. Xstream consists of over 24 KLOC and its code is accompanied by a set of test cases for each version. Xstream comes with many non-trivial test cases that parse complex XML files, thus being an interesting case for comparison between our selection of case studies and makes this project more challenging to investigate.

[5] http://joda-time.sourceforge.net/
[6] http://barbecue.sourceforge.net/
[7] http://pmd.sourceforge.net/
[8] http://www.oracle.com/technetwork/developer-tools/jdev
[9] http://www.eclipse.org
[10] http://www.jedit.org/
[11] http://www.jetbrains.com/idea/
[12] http://ant.apache.org/
[13] http://www.gnu.org/software/emacs/
[14] http://xstream.codehaus.org/

7.3 Applicability of *Test Reuse Patterns*

This section addresses *RQ1*: *How frequently can Test Reuse Patterns be applied to evolve test cases?* Our empirical study of several source repositories in popular open-source projects indicates that the test cases repaired and produced with our *Test Reuse Patterns* can cover a significant part of the software systems. The study suggests that *Test Reuse Patterns* are more effective when applied to more mature software systems than in the initial stage of the software lifecycle.

7.3.1 Change of Method Declaration

To determine the potential applicability of "*change of method declaration*" pattern, we measured how often software developers modify method signatures with respect to the overall number of changes that may lead to compilation errors in the test cases.

We collected 162 versions belonging to the 5 subjects introduced in Section 7.2. For each consecutive version we counted the number of changes and the number of signature changes among them. Table 7.2 shows the results of our analysis. Column *Subject* shows the subject name; Column *Versions* shows the number of versions analyzed for each project.

We used *JDiff* to identify the changes between two consecutive releases of each project, and counted the number of changes that can lead to compilation errors in test cases.

Column *Error raising Changes* shows the total amount of changes that may lead to compilation errors in the test cases. Column *TCA* indicates the number of changes in which TCA can be applied to repair the compilation errors. TCA can handle a total of 2,146 changes: 890 *added parameters*, 663 *removed parameters*, 319 *parameter type changes*, and 274 *return type changes*. Table 7.3 shows the distribution of changes in the method declaration across 5 open-source projects. On average, TCA can be applied to repair more than 40%[15] of the changes that lead to compilation errors.

These results indicate that the changes in parameter declarations are frequent and that TCA can be applied to many of them. Since the test cases are not available for all the changed methods, we did not compute the exact number of test cases that raise compilation errors but the number of changes that might cause these errors.

[15]The percentage is calculated using macro average formula that shows the average of changes that TCA can handle over all changes that raise compilation error regardless of the software projects.

Subject	Versions	TCA	Error raising Changes	%
Xstream	22	175	819	21.37
PMD	60	508	2,477	20.51
Barbecue	2	2	2	100.00
JodaTime	30	889	2,000	44.45
JfreeChart	48	572	2,976	19.22

Table 7.2. Applicability of TCA on open source projects

Subjects	Number of Parameter Type Change	Number of Parameter Add	Number of Parameter Remove	Number of Return Type Change
Xstream	12	91	61	11
PMD	112	168	144	84
Barbecue	0	1	1	0
JodaTime	93	366	321	109
JfreeChart	102	264	136	59
Total	319	890	663	274

Table 7.3. Amount of changes supported by TCA across projects

7.3.2 Extension of Hierarchy and Implementation of Interface

To evaluate the applicability of TCA for generating test cases we computed the average number of classes belonging to each subject for which TCA can generate test cases. To identify the number of classes for which TCA can be applied, we identified the number of classes that either belong to a class hierarchy of project or implement an interface.

Table 7.4 shows the results: Column *Subject* indicates the version of the subject that we investigated, Column *Classes* indicates the total amount of testable classes in the analyzed version, Column *Interface* indicates the number and percentage of classes that implement an interface, Column *Hierarchy* indicates the number and percentage of classes that extend a class hierarchy, i.e., classes for which TCA could generate test cases. Column *In+Hi* shows the number and percentage of classes that either implement an interface or belong to a class hierarchy.

According to our results the "*extension of class hierarchy*" *Test Reuse Pattern* is applicable to 794 out of 1,249 classes, 63.6% of the cases, while the "*implemen-*

Subject	Classes	Interface	Hierarchy	In+Hi
Xstream 1.31	185	24(14.1%)	103(55.7%)	127(68.6%)
PMD 4.2	472	36(7.6%)	315(66.7%)	342(72.5%)
Barbecue 1.5b1	49	3(6.1%)	27(55.1%)	30(61.2%)
JodaTime 1.62	97	32(33.0%)	74(76.3%)	77(79.4%)
JFreeChart 1.013	446	322(72.2%)	275(61.7%)	392(87.9%)
Average		33.5	63.6	77.5

Table 7.4. Applicability on Test Generation for Class Hierarchies and Interface

tation of interface" *Test Reuse Pattern* is applicable to 419 classes (33.5%). The data also shows that when the two *Test Reuse Patterns* are combined together TCA can generate test cases for 968 classes (77.5%).

The applicability of the "*extension of class hierarchy*" *Test Reuse Pattern* is uniform in all projects, in fact, the standard deviation is 8.7, which suggests that the "*extension of class hierarchy*" could be successfully applied on different subjects too.

On the other hand, the applicability of the "*implementation of interface*" *Test Reuse Pattern* is less uniform, the standard deviation of this pattern is 27.64. This depend on the fact that in 2 of the 5 subjects, Barbecue and PMD, the *Interface to class ratio* (the total number of interfaces divided by the total number of classes [SSK03]) is 5.4% (3/55) and 10.5% (51/483) respectively. This shows that in Barbecue and PMD interfaces are under utilized in comparison to common software systems (a study of Steimann et al. [SSK03] shows that this ratio on average is 25%).

The results presented in this section represent an upper bound to the applicability of the approach, in fact the applicability of TCA depends not only on the presence of class hierarchies but also on the availability of test cases for classes of the same hierarchy. The applicability of the algorithm depends on the number of test cases available for existing classes in the hierarchy. The applicability of TCA benefits from the presence of several classes that extend hierarchies, which are high (60%) in the projects considered. In practice the applicability of TCA may be lower. For example, in JodaTime 1.62, TCA could generate test cases for 76% (74/97) of the classes, but JodaTime does not have test cases for all the implemented classes, thus TCA can automatically generate test cases for 32% (32/97) of classes.

Subjects	Methods	Override	Overload
Xstream 1.31	1094	185(16.9%)	572(52.3%)
PMD 4.2	2443	280(11.5%)	1261(51.6%)
Barbecue 1.5b1	223	11(4.9%)	86(38.6%)
JodaTime 1.62	2053	329(16%)	1373(66.9%)
JFreeChart 1.013	6175	641(10.4%)	3329(53.9%)
Average		12.1%	55.2%

Table 7.5. Applicability of TCA on Override/Overload *Test Reuse Pattern*

7.3.3 Introduction of Overriding and Overloading Methods

In this section we discuss the applicability of the "introduction of overridden and overloaded method" *Test Reuse Patterns*. To this end, we measured the number of methods in the subject programs for which TCA can be applied. Table 7.5 shows the results. Column *Methods* shows the number of public methods in the project. Column *Override* shows the number of overridden methods. Column *Overload* shows the number of methods on which the "*introduction of overloaded method*" *Test Reuse Pattern* can be applied, i.e., pure overloaded methods and methods that have the same name of a method declared in another class belonging to the same hierarchy. On average the "*introduction of overloaded method*" *Test Reuse Pattern* can be applied to slightly over the half of the visible methods of the classes in the projects, 55.2%. While the "*introduction of overridden method*" *Test Reuse Pattern* can be applied just to 12.1% of cases.

7.4 Effectiveness of Change of Method Declaration

This section answers **RQ2**: *How effective are the test cases adapted by TCA? How do these test cases compare with the test cases generated by state-of-the-art techniques?* specifically for the "*change of method declaration*" *Test Reuse Pattern*.

We applied TCA to repair 138 test cases of the subjects of our study. We considered 6 releases and 21 test cases for JFreeChart, 2 releases and 18 test cases for JodaTime, 13 releases and 99 test cases for PMD. For each release of the software, we executed TCA on all test cases that do not compile. The test cases considered in our study were broken by different type of changes: parameter type changes (26), parameter additions (68), parameter removals (23), and return type changes (21). Table 7.6 shows the obtained results. Column *Errors-fixed*

shows the number of test cases whose compilation errors have been corrected by TCA. TCA correctly repairs all the compilation errors. Column *Valid-test-cases* shows instead the number of the test cases that do not show any runtime failure after repair, which is 128 (92.75%).

The 10 failures of the repaired test cases depend on the default values generated by TCA that alter the test behavior, thus causing failures in assertions (4 cases), or runtime exceptions due to the absence of proper initialization for other variables (6 cases). The column *Same-as-Developers* shows the number of test cases that TCA corrected as done by developers. To this end, we compared the test cases repaired by TCA with the ones repaired by software developers to check if they present the same behavior. Of the 128 repaired test cases, 105 present the same behavior of the test cases generated by software developers, while the others present a different but valid behavior. Similar behavior is measured by counting instructions that both test cases cover during execution.

Change	TC	Errors fixed	Valid test cases	Same as Developers
Parameter Type Change	26	26	22 (84.62%)	14 (53.85%)
Parameter Add	68	68	68 (100.00%)	59 (86.76%)
Parameter Remove	23	23	17 (73.91%)	21 (91.03%)
Return Type Change	21	21	21 (100.00%)	11 (52.38%)
Total	138	138	128 (92.75%)	105 (76.08%)

Table 7.6. Effectiveness of Generating Repairs

We compared the variables (either objects, fields or primitive variables) initialized by software developers with the variables initialized by TCA to determine the distance of the repairs of TCA from the ones suggested by the developers. Table 7.7 shows the results. Software developers initialized 124 different variables (column *Devels*), while TCA initialized 119 variables (column *TCA-All*), and thus TCA automatically initializes more than 95% of the variables manually initialized by the developers. The variables that TCA fails to initialize are variables required for the execution of methods that do not present changes in parameters declarations. TCA focuses on methods that present changes in parameters declaration only, thus cannot determine that it is necessary to initialize input variables or object fields to properly execute methods that present changes in their functionality.

Column *TCA-Correct* reports the number of variables properly initialized by TCA. We consider a value as properly initialized if either its value is the same as

		Variables Initialized				
Change	TC	Devels	TCA-All	TCA-Correct	Pr	Re
Parameter Type Change	26	31	26	14	0.54	0.45
Parameter Add	68	66	68	59	0.87	0.89
Parameter Remove	23	6	5	5	1.00	0.83
Return Type Change	21	21	20	20	1.00	0.95
Total	138	124	119	98	0.85	0.78

Table 7.7. Effectiveness of Finding Initialization Values

the one used by the developers, or the value is different but neither the execution path of the software nor the generated results are different. We use these data to compute the precision and the recall of TCA in initializing program variables (columns *Pr* and *Re*). We calculate the *precision* as the fraction of the variables initialized by TCA that are correctly initialized, and the *recall* as the fraction of the variables initialized by software developers that are also properly initialized by TCA.

The overall precision (0.85) is high. The lowest precision corresponds to test cases affected by changes in the parameter type of the methods. This depends on the fact that changes in parameter type often correspond to changes in the structure of the methods that include many simultaneous modifications, renamed variables, new variables, loops, etc. Many simultaneous changes in the same method reduce the efficacy of function `corresponding` (defined in Section 4.4.1) that currently combines *Unix diff*, *Levenshtein algorithm*, and *strings alignment*. The presence of multiple changes in the code limits the ability of identifying the corresponding lines because of many mismatches in the compared lines. The overall recall (0.78) is also high. The high precision indicates that although TCA cannot identify all the variables, it tends to initialize the variables with proper values.

7.5 Effectiveness of Generating Tests for New Classes

In this section we answer to **RQ2** for the "*extension of class hierarchy*" and "*implementation of interface*" *Test Reuse Patterns* based on two measures: code coverage and conciseness.

Subjects	C	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	36	**58.55**	78.19	18.67	54.18	43.35
PMD 4.2	56	**47.18**	65.53	40.56	49.68	60.03
Barbecue 1.5	12	**73.42**	63.44	45.80	23.10	82.03
JodaTime 1.62	32	**75.28**	88.07	48.97	77.33	71.52
Jfrechart 1.013	204	**47.92**	49.22	27.70	52.06	51.46
Average		**60.47**	68.89	36.34	51.27	61.68

Table 7.8. Effectiveness of Test Generation for *"extension of class hierarchy" Test Reuse Pattern*

7.5.1 Code Coverage

We evaluated the effectiveness of TCA by generating test cases for all the classes that belong to a class hierarchy in the five subjects of the study. For each class considered in the experiment, we removed the test cases implemented by developers for that class, and applied TCA to generate new test cases.

Table 7.8 shows the results on 340 classes: Column *C* indicates the number of classes for which TCA generated the test cases. The other columns indicate the instruction coverage for the class under test obtained with the test cases generated with the different approaches.

The test cases produced by the developers obtain the highest coverage, but at a price of a high effort. TCA performs as good as EvoSuite, and outperforms both CodePro and Randoop. To characterize the differences between TCA, developers, and EvoSuite, we compared the set of instructions covered by test cases generated by developers, TCA, and EvoSuite for each class under test. The test cases generated by TCA and EvoSuite cover a common set of 12055 instruction of code (49% of the total). TCA and EvoSuite are complementary: TCA covers 6370 instructions (25.9%) not covered by EvoSuite, while EvoSuite covers 6208 instructions not covered by TCA (25.2%). TCA outperforms EvoSuite in 130 test cases, EvoSuite outperforms TCA in 134. EvoSuite works better than TCA when path conditions cannot be covered by copying data used in existing test cases. On the other hand, TCA produces test cases that covers some parts of the source code that can only be covered with domain knowledge that TCA implicitly imports from existing test cases.

The test cases produced by developers often stress both the code of the class under test and the code of parent classes, to identify integration faults between parent and child classes. Table 7.8 shows the coverage for the class under test

Subjects	C	In	In+Hi	Dev	Randoop	CodePro	Evo
Xstream 1.31	26	44.96	82.42	94.12	45.42	80.31	51.32
PMD 4.2	18	69.44	70.81	65.72	21.28	43.17	59.83
Barbecue 1.5b1	3	100	100	90.91	77.67	100	89.00
JodaTime 1.62	31	35.91	56.94	93.12	74.64	94.33	54.48
JfreeChart 1.013	309	14.44	55.68	64.61	34.11	77.49	40.09
Average		**62.58**	**77.54**	85.97	54.75	79.45	63.66

Table 7.9. Effectiveness on Test Generation for "implementation of interface" *Test Reuse Pattern*

only. We manually inspected the generated test cases and found that by reusing existing test cases, TCA generates test cases that check both the class under test and its integration with parents. EvoSuite is configured to generate test cases for single classes and produces good test cases for that class, but ignores the integration with the parent class. For example, for the class `EthiopicChronology`, TCA covers 1083 statements that belong to class `EthiopicChronology` or one of its parents, while EvoSuite covers only 683 statements.

By modifying existing test cases produced by the developers, TCA generates test cases that are more likely to be readable than the ones produced by EvoSuite. Figure 5.2 shows a test case generated by TCA that although covers a complex behavior is still easy to understand thanks to the presence of meaningful names. Listing 7.1 shows a test case generated by EvoSuite that uses abstract names that make test cases difficult to be understood.

```
1 public void test2()  {
2    EthiopicChronology var0 = EthiopicChronology.getInstanceUTC()
        ;
3    assertNotNull(var0);
4    DateTimeField var1 = (DateTimeField)var0.weekyearOfCentury();
5    long var2 = var1.addWrapField(-803L, 65533);
6    assertEquals(var2, 1041465599197L);
7 }
```

Listing 7.1. A test case for class `EthipicChronology` by EvoSuite.

Table 7.9 shows the results of the instructions covered by "*implementation of interface*" pattern and its comparison to other techniques on 387 classes. Column *In* shows the results of "*implementation of interface*" pattern which TCA outperforms Randoop and is comparable with results of EvoSuite (column *Evo*). However, the test cases generated by CodePro outperform other automated tech-

Subjects	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	**2.81**	5.53	667.53	10.39	5.00
PMD 4.2	**3.41**	9.23	316.86	12.21	6.54
Barbecue 1.5b1	**18.67**	27.58	250.50	33.75	15.67
JodaTime 1.62	**5.19**	5.81	344.22	42.75	19.56
JfreeChart 1.013	**7.19**	7.27	927.79	39.78	19.73
Average	**7.45**	11.09	501.38	27.78	13.30

Table 7.10. Average number of test cases per class for *"extension of class hierarchy" Test Reuse Pattern*

niques. We applied both "*extension of class hierarchy*" and "*implementation of interface*" patterns to see how our results get improved. The instruction coverage obtained with applying both patterns (column *In+Hi*) shows that TCA can cover 15% more instructions than just applying "*implementation of interface*" pattern. Moreover, the results are comparable with CodePro which is the best state-of-the-art tool in this category of classes. Our investigation on "*implementation of interface*" pattern shows that this *Test Reuse Pattern* is not as effective as "*extension of class hierarchy*" pattern but can be used where "*extension of class hierarchy*" is not applicable i.e., no sibling classes are available or sibling classes are not associated with many test cases.

7.5.2 Conciseness

We measured the number of test cases generated by TCA, developers, and state-of-the-art tools. To this end, we counted the number of test methods in the test classes of class under test. As shown in Tables 7.10 and 7.11, we collected these data from five case studies for two *Test Reuse Patterns*. Columns 2-6 show the average number of test methods generated by TCA, developers, Randoop, CodePro, and EvoSuite, respectively.

To obtain these results we counted the number of test methods that exercise the class under test and discarded helpers, setup, teardown methods. As the results suggest, on average, the number of test cases generated by TCA is lower than the number of test cases generated by the other approaches, even than the number of test cases written by the developers in "*extension of class hierarchy*" pattern (Table 7.10). TCA generates 33% fewer test cases that the developers, and 43% fewer test cases than the best automated tool (EvoSuite).

Table 7.11 shows the average size of test suite generated for classes in the

Subjects	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	**1.35**	3.12	20.88	7.15	3.12
PMD 4.2	**1.94**	1.33	891.72	10.72	2.56
Barbecue 1.5b1	**2.00**	2.33	77.67	2.00	2.00
JodaTime 1.62	**1.33**	1.33	257.09	7.85	2.55
JFreeChart 1.013	**5.29**	5.29	691.20	22.37	17.68
Average	**1.66**	2.03	417.42	6.93	2.55

Table 7.11. Average number of test cases per class for *"implementation of interface" Test Reuse Pattern*

"*implementation of interface*" *Test Reuse Pattern*. The results show that TCA generates fewer test cases than the other approaches, even than the test cases written by the developers. These results are consistent with the "*extension of class hierarchy*" *Test Reuse Pattern*.

7.6 Effectiveness of Generating Tests for New Methods

In this section we answer **RQ2** for the "*introduction of overloaded method*" and the "*introduction of overridden method*" *Test Reuse Patterns* based on two measures: code coverage and conciseness.

7.6.1 Code Coverage

We evaluated the ability of TCA to adapt test cases for new methods by applying TCA to all the testable methods in the considered case studies. Tables 7.12 and 7.13 show the results in terms of statement coverage for the "introduction of overloaded and the overridden method" *Test Reuse Patterns*.

Table 7.12 presents the comparison of the code coverage of the test cases generated by applying the "*introduction of overloaded method*" pattern, developers, Randoop, CodePro, and EvoSuite. The rows in the table show results for each case study. The column (*M*) shows the number of methods under test in each case study. Out of 2319 methods, TCA achieved an average coverage of 71% which is better than all the automated test generation tools. TCA performed worse than CodePro and EvoSuite only on the JodaTime project.

Subjects	M	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	177	74.75	83.85	30.09	67.63	54.31
PMD 4.2	46	71.20	79.39	40.85	67.04	61.11
Barbecue 1.5b1	47	78.41	59.63	51.78	45.80	70.20
JodaTime 1.62	523	57.76	75.16	35.82	76.81	62.63
JFreeChart 1.013	1526	73.35	72.14	54.79	80.40	75.58
Average		71.09	74.04	42.67	67.53	64.77

Table 7.12. Effectiveness of *"introduction of overloaded method" Test Reuse Pattern*

Subjects	M	TCA	Developer's	Randoop	CodePro	EvoSuite
Xstream 1.31	5	30.40	82.00	6.00	44.80	26.60
Barbecue 1.5b1	7	81.00	92.86	41.71	70.29	100.00
JodaTime 1.62	10	84.00	100.00	96.00	93.30	74.90
JFreeChart 1.013	121	55.50	70.61	47.90	82.96	84.16
Average		62.73	86.37	47.90	72.84	71.41

Table 7.13. Effectiveness on *"introduction of overridden method" Test Reuse Pattern*

Table 7.13 presents the code coverage results of the test cases generated by applying the "*introduction of overridden method*" *Test Reuse Pattern* on four case studies. TCA covers a substantial part of the methods under tests, however due to lack of test cases for the original methods, the results of coverage of test cases generated by TCA are not as good as other test patterns.

7.6.2 Conciseness

To show the conciseness of the adapted test cases we count the number of generated test cases for each method under test. Tables 7.14 and 7.15 show the average number of test cases generated for the "introduction of overloaded and overridden method" *Test Reuse Patterns*. As Table 7.14 suggests TCA generates fewer test cases in comparison with the test cases generated by the developers and by other automated tools. In fact, TCA generates 35% fewer test cases than the developers on average.

Table 7.15 presents the average number of test cases generated by the differ-

Subjects	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	**3.43**	5.44	606.25	3.65	0.95
PMD 4.2	**1.22**	2.7	1057.96	2.91	2.33
Barbecue 1.5b1	**1.22**	3.53	701.29	2.86	1.29
JodaTime 1.62	**3.29**	5.01	740.26	3.45	1.67
JFreeChart 1.013	**3.01**	2.22	2622.13	15.61	6.41
Average	**2.43**	3.78	1145.58	5.7	2.53

Table 7.14. Average number of test cases generated with *"introduction of overloaded method" Test Reuse Pattern*

Subjects	TCA	Developer	Randoop	CodePro	EvoSuite
Xstream 1.31	**1.00**	7.6	3560.8	7.4	1.4
Barbecue 1.5b1	**1.86**	2.57	1026.29	9.29	0.86
JodaTime 1.62	**1.00**	4.4	1123.5	2.4	1
JFreeChart 1.013	**1.20**	5.79	1210.25	13.74	7.05
Average	**1.26**	5.09	1730.21	8.21	2.58

Table 7.15. Average number of test cases generated with *"introduction of overridden method" Test Reuse Pattern*

ent approaches, and shows that TCA generates less test cases than the developers and other automated tools, for the "*introduction of overridden method*" *Test Reuse Pattern*. TCA generates 75% fewer test cases than the developers and 51% fewer than best automated tool for this *Test Reuse Pattern*.

7.7 Discussion

We inspected the results of TCA on adapting and generating test cases to identify the advantages and disadvantages of our technique. We discuss most common problems that we found in generated test cases in our case studies.

7.7.1 Availability of Test Cases to Reuse

TCA generates the test cases essentially by reusing existing test cases. Limited availability of test cases in the repository of projects directly affects the effectiveness of TCA.

For example, to generate test case for `ISO8601GregorianCalendarConverter`, a new class in JodaTime, TCA uses test cases from 9 sibling classes. However, the test cases of three of the sibling classes do not cover any part of the class under test, the test classes of three of the sibling classes cover 100% of new class `ISO8601GregorianCalendarConverter`, and the test cases of the other three classes are redundant.

```
1     public boolean canConvert(Class type) {
2         return type.equals(GregorianCalendar.class);
3     }
4
5     public Object fromString(String str) {
6         for (int i = 0; i < formattersUTC.length; i++) {
7             DateTimeFormatter formatter = formattersUTC[i];
8             try {
9                 DateTime dt = formatter.parseDateTime(str);
10                Calendar calendar = dt.toCalendar(Locale.
                      getDefault());
11                calendar.setTimeZone(TimeZone.getDefault());
12                return calendar;
13            } catch (IllegalArgumentException e) {
14                // try with next formatter
15            }
16        }
17        String timeZoneID = TimeZone.getDefault().getID();
18        for (int i = 0; i < formattersNoUTC.length; i++) {
19            try {
20                DateTimeFormatter formatter = formattersNoUTC[i
                      ].withZone(DateTimeZone.forID(timeZoneID));
21                DateTime dt = formatter.parseDateTime(str);
22                Calendar calendar = dt.toCalendar(Locale.
                      getDefault());
23                calendar.setTimeZone(TimeZone.getDefault());
24                return calendar;
25            } catch (IllegalArgumentException e) {
26                // try with next formatter
27            }
28        }
29        throw new ConversionException("Cannot parse date " + str
              );
30    }
31
32    public String toString(Object obj) {
```

```
33          DateTime dt = new DateTime(obj);
34          return dt.toString(formattersUTC[0]);
35      }
```

Listing 7.2. Class `ISO8601GregorianCalendarConverter`

Listing 7.2 shows the body of the new class `ISO8601GregorianCalendarConverter` that is composed of three methods `canConvert`, `fromString`, and `toString`. The test cases adapted from three sibling classes `DateConverter`, `ISO8601DateConverter`, and `StringConverter` are complementary. In fact, test cases adapted from class `DateConverter` covers `toString` method and lines 6-18 of method `fromString`. Lines 20-24 of method `fromString` are covered with test cases adapted from class `ISO8601DateConverter`. Method `canConvert` is covered with the test cases adapted from class `ToStringConverter`. This example shows that we need a certain number of test cases to be able to generate effective test cases. In this case, test cases from three sibling classes are enough to cover all lines of the class under test.

7.7.2 Using Mock Objects

In the process of adapting test cases, wherever we could not find any reusable inputs, we use mock objects to initialize the parameters. However, since mock objects do not provide any implementation, the test case might fail to execute. We present two examples and possible solution to avoid this problem.

```
1   @Test
2    public void testIsExceptionBlockParameter() {
3        ASTTryStatement tryNode = new ASTTryStatement(1);
4        ASTBlock block = new ASTBlock(2);
5        ASTVariableDeclaratorId v = new ASTVariableDeclaratorId
             (3);
6        v.jjtSetParent(block);
7        block.jjtSetParent(tryNode);
8        assertTrue(v.isExceptionBlockParameter());
9    }
```

Listing 7.3. Test Case of developers for class `ASTVariableDeclaratorId`

The listing 7.4 presents an example of TCA test case that is generated to cover constructor `ASTVariableDeclaratorId(JavaParser p, int i)` of class `ASTVariableDeclaratorId` in PMD 4.2. TCA uses the test cases written for another constructor in a previous version of the application. The Listing 7.3 shows the original test case written for the constructor `ASTVariableDeclaratorId(int i)`.

Line 5 instantiates an object of type `ASTVariableDeclaratorId` by calling constructor with an integer parameter.

The new overloaded constructor accepts two parameters: the first parameter is of type `JavaParser` and the second parameter is an integer. The second parameter is reused when adapting the test case (Listing 7.3) and is replaced in the constructor call. However, TCA can not find any variable for first parameter. Thus TCA calls the constructor of class `JavaParser` that accepts one input of type `CharStream`. TCA uses the mock objects to initialize `CharStream` object, as it is shown in Line 5. Although the generated test case compiles, it raises an exception complaining about the missing definition of the preceding method call that is not implemented in the mock object.

```
1  @Test
2  public void testOverload_ASTVariableDeclaratorId_Q...() {
3    ASTTryStatement tryNode = new ASTTryStatement(1);
4    ASTBlock block = new ASTBlock(2);
5    ASTVariableDeclaratorId v = new ASTVariableDeclaratorId(
6             new net.sourceforge.pmd.ast.JavaParser(org.easymock.
                 EasyMock.createMock(CharStream.class)), 3);
7    v.jjtSetParent(block);
8    block.jjtSetParent(tryNode);
9    assertTrue(v.isExceptionBlockParameter());
10 }
```

Listing 7.4. A TCA test case that uses mock objects

```
1  @Test
2  public void testOverload_ASTVariableDeclaratorId_Q...() {
3    ASTTryStatement tryNode = new ASTTryStatement(1);
4    ASTBlock block = new ASTBlock(2);
5    ASTVariableDeclaratorId v = new ASTVariableDeclaratorId(
6             TargetJDKVersion.DEFAULT_JDK_VERSION.createParser(new
                  StringReader("")), 3);
7    v.jjtSetParent(block);
8    block.jjtSetParent(tryNode);
9    assertTrue(v.isExceptionBlockParameter());
10 }
```

Listing 7.5. A manually repaired test case of TCA

These test cases are not considered valid for the new overloaded method. However, the developer only needs to fix the first parameter of the constructor call and the test case will run on the modified software. One way to solve this problem is to see how the developers initialized any variable of type `JavaParser`.

A quick look at the test repository of PMD project shows that the developers use a factory method to instantiate the `JavaParser` class. By replacing the first parameter of the `ASTVariableDeclaratorId` constructor with the factory method, the test case runs smoothly on the new overloaded method. The modified test case is shown in Listing 7.5.

Chapter 8

Conclusion

Testing software systems is the most popular way of verifying software systems. Since software evolves frequently, developers need to constantly update test cases. This process is time consuming and error-prone, thus automating the test evolution process can reduce the overall cost of software evolution. The problem of evolving test cases recently gained considerable attention in the research community. So far, researchers have mainly focused on regression testing [HO08, XKK+10], repairing failing test cases [DJDM09, DGM10], and repairing GUI test cases [Mem08, FGX09].

This dissertation investigates the problem of test suite evolution by proposing a new approach to repair and evolve test cases. We empirically studied the evolution of test cases in test repository, and noticed some recurrent patterns in the structure of test cases written by developers. We exploited this redundancy to define a set of algorithms to automatically generate and repair test cases to verify new or modified functionality of software systems.

We refer to the common activities that the developers follow to write similar test cases as *Test Reuse Patterns*. We identified five different *Test Reuse Patterns*: change of method declaration, extension of class hierarchy, implementation of interface, introduction of overload and override methods. We noticed that when the declaration of a method changes, specifically by adding a parameter, changing a parameter type, changing a return type or removing a parameter, software developers tend to repair the test cases that do not compile any more by using their domain knowledge to resolve compilation errors and update the obsolete test cases. We also notice that when a software system is extended with a new class, developers tend to reuse the test cases of sibling classes or classes that implement the same or similar interfaces to generate test cases for the new class. Moreover, when developers add a new method to a class, they tend to reuse the

test cases written for overloading and overriding methods to generate test cases for the new methods.

We defined algorithms that automate the *Test Reuse Patterns* that we identified so far, thus reducing the effort and time required to write new test cases. We propose a framework, TCA, that repairs test cases that do not compile due to changes in the software system, and that generates test cases when new classes and methods are added to the system. To repair not compiling test cases, TCA first finds the type of change. According to the type of change, TCA finds initialization variables and their corresponding values by combining dynamic and static data flow analysis. Then TCA proposes a repair to the developer which preserves the behavior of the test case. To generate test cases for new elements of the system, TCA extracts similar test cases that can be used to generate test cases from the repository of software system, then TCA adapts the test cases by updating the references, resolving compilation errors, repairing failing test cases, and removing redundant test cases.

We evaluated the potential applicability and effectiveness of the proposed framework by experimenting on five open-source projects. We evaluated the applicability of our framework by examining the amount of test cases that TCA can potentially repair and generate. We evaluated the effectiveness of the approach by comparing the code coverage obtained with the test cases generated by TCA with the coverage obtained with the test cases written by the developers and generated with the state-of-the-art test generators like Evosuite, Randoop, and CodePro. Our evaluation results suggest that TCA can effectively be used by software developers to reduce the effort of maintaining test suites.

8.1 Contributions

The first contribution of this dissertation is the introduction of *Test Reuse Patterns* which to the best of our knowledge have not been identified in the literature. This dissertation provides an empirical study of the redundancy of test cases in software test suites, and proposes possible ways to exploit this redundancy.

The second major contribution of this dissertation is the development of the TCA framework, which exploits *Test Reuse Patterns* to repair and evolve test cases in software systems.

We now summarize in more details some aspects of these two major contributions:

Empirical study of test suite repositories. We analyzed the test suite repositories of multiple versions of several software projects to identify the sim-

ilarity of modified and added test cases with already available test cases. As a result, we identified some changes that guide developers in writing test cases (Chapter 3).

Test Reuse Patterns. We identified the changes that guide software developers in writing test cases as *Test Reuse Patterns* that represent common practice of developers who reuse and adapt test cases to evolve exiting test suites. The *Test Reuse Patterns* that we identified include: change of method declaration, extension of class hierarchy, implementation of interface, introduction of overloaded and overridden methods (Chapter 3). The results have been published in [Mir11, MPP12].

Test suite evolution framework. We proposed a general framework to both repair obsolete test cases and generate new test cases by reusing existing ones. Our framework exploits the *Test Reuse Patterns* introduced in this dissertation and is generally applicable to new *Test Reuse Patterns* (Chapter 3). The results have been published in [MPP12].

Test suite repair. We presented a technique to repair the test cases that do not compile any more due to changes in some method declarations. Our technique leverages data flow analysis and dynamic instrumentation to suggest developers how to repair the test cases in a way that preserves the behavior of the test cases after the modification (Chapter 4). The results have been published in [MPP10, MP11, MPP11, MPP12].

Test suite adaptation. We proposed a technique to generate test cases for new classes and methods by reusing existing test cases and adapting them for the new elements. The adaptation technique finds similar test cases, updates the references to new elements, resolves compilation errors, repairs failing test cases, and removes redundant test cases. The technique enables the reuse of existing test cases to test new functionality instead of writing the test cases from scratch (Chapter 5). The results have been published in [MPP12].

Prototype implementation. We implemented our framework in an prototype tool called TCA that implements the techniques for repairing and generating test cases. We used TCA prototype to experimentally evaluate the effectiveness of our technique on real world software projects (Chapter 6). The results have been published in [MP11, MPP12].

Framework evaluation. We evaluated the test adaptation framework by applying TCA to five open-source projects. The results of our experiment shows that TCA is applicable for testing a fair portion of software systems. We evaluated the effectiveness of our framework by comparing the repairs suggested by TCA on 138 test cases with the repairs suggested by developers. The comparison shows the viability of our approach. We applied TCA to adapt test cases of more than 700 classes and 2400 methods of five open-source projects. This study shows that our technique can generate test cases that achieve a coverage comparable with the coverage obtained with automated test generation tools. We also verify that our technique complements the test cases generated by automated tools since TCA can cover part of code that test cases produced by automated tools can not cover and vice versa (Chapter 7). The results have been published in [MPP12].

8.2 Future Research Directions

The idea of *Test Reuse Patterns* and the framework that automates the process of repairing and generating test cases open several new research directions:

Extending TCA with test generation tools: Although our results are comparable with test generation techniques by means of coverage, we can improve the coverage results of our framework by extending the framework with dynamic symbolic execution (DSE) engines or search based test (SBST) input generators. DSE and SBST can help TCA improve the code coverage by generating test inputs that TCA can not produce. For example in the presence of a method with a new sets of parameters, TCA can ask a DSE or SBST tool to generate some possible inputs that increase the code coverage. It would be interesting to explore the possibility of integrating TCA with JavaPathFinder [VPK04] and EvoSuite.

Improving the applicability of TCA: We identified an initial set of interesting *Test Reuse Patterns* thus opening the research for new *Test Reuse Patterns*.

Generalizing test cases: Another issue that we need to address is copy and pasting test cases which is an anti-pattern. We investigate the ways to factorize the test cases by producing more maintainable test cases using object oriented design principles like encapsulation.

Readability and understandability of TCA test cases: One of the advantages of TCA is to adapt test cases that are already written by the developers, thus the test cases will be more understandable than test cases generated by automated tools. We manually checked some of the test cases and identified that the test cases by TCA are closer to actual test scenarios which the developers are interested. However, an empirical study with some experienced users (i.e., real developers) is required to evaluate the actual readability and understandability of the test cases.

Porting to Web application test evolution: This dissertation developed the concept of *Test Reuse Patterns* for desktop applications. We think that the technique is not bound to this domain, and our long-term plan is to adapt and generalize the technique to a wider class of applications, such as web applications. To generalize TCA to the web applications, we would have to first examine how current *Test Reuse Patterns* fit into web application testing context and find some new patterns that are specialized in web application testing. Dynamic, heterogeneous, and distributed nature of web applications would make this task more challenging.

Bibliography

[ALM07] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: a two-level genetic-random unit test data generator. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 144–153, 2007.

[AM09] James H. Andrews and Tim Menzies. On the value of combining feature subset selection with genetic algorithms: faster learning of coverage models. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–10, 2009.

[AML10] James H. Andrews, Tim Menzies, and Felix C.H. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 99:80–94, 2010.

[ASC+06] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. *Academic and Industrial Conference on Practice And Research Techniques, Testing*, pages 137–146, 2006.

[Bal98a] Thomas Ball. On the limit of control flow analysis for regression test selection. *SIGSOFT Software Engineering Notes*, 23:134–142, 1998.

[Bal98b] Thomas Ball. On the limit of control flow analysis for regression test selection. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 134–142, 1998.

[Bin97] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498 –516, 1997.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 123–133, 2002.

[BLH09] Lionel Briand, Yvan Labiche, and Siyuan He. Automating regression test selection based on UML designs. *Information and Software Technology*, 51(1):16–30, 2009.

[BLO03] Lionel Claude Briand, Y Labiche, and L O'Sullivan. Impact analysis and change management of UML models. In *Proceedings of the International Conference on Software Maintenance*, pages 256–265, 2003.

[BM07] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. In *DOSTA '07: Workshop on Domain specific approaches to software test automation: in conjunction with the 6th Foundations of Software Engineering Conference*, pages 1–7, 2007.

[BM10] Luciano Baresi and Matteo Miraz. TestFul: automatic unit-test generation for Java classes. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 281–284, 2010.

[BWK05] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 571–579, 2005.

[BZ11] Martin Burger and Andreas Zeller. Minimizing reproduction of software failures. In *ISSTA '11: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 221–231, 2011.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83:60–66, 2010.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 71–80, 2008.

[CM05] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 309–312, 2005.

[CM07] Tsong Yueh Chen and R. Merkel. Quasi-random testing. *IEEE Transactions on Reliability*, 56(3):562 –568, 2007.

[Cod12] Google CodePro. *http://code.google.com/javadevtools/codepro/doc/index.html*, accessed August 2012.

[CPU02] Yanping Chen, Robert L. Probert, and Hasan Ural. Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 214 – 223, 2002.

[CPU07] Yanping Chen, Robert L. Probert, and Hasan Ural. Model-based regression test suite generation using dependence analysis. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 54–62, 2007.

[CRV94] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. TESTTUBE: a system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211 –220, 1994.

[CYK+05] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin Sullivan. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, 2005.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Object-Oriented Programming Conference*, pages 404–428, 2006.

[DGM10] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In *ISSTA '10: 2010 International Symposium on Software Testing and Analysis*, pages 207–218, 2010.

[DJDM09] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the 24th IEEE/ACM international Conference on Automated software engineering*, pages 433–444, 2009.

[DLM+11] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. Automated GUI refactoring and test script repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 38–41, 2011.

[dMB08] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340, 2008.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 397–407, 2009.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438 –444, 1984.

[DNMJ08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 441–450, 2008.

[DNSVT07] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, 2007.

[dPX+06] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 59–68, 2006.

[DR12] Barthélémy Dagenais and Martin P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 47–57, 2012.

[FA11a] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Symposium on the Foundations of Software Engineering*, pages 416–419, 2011.

[FA11b] Gordon Fraser and Andrea Arcuri. Whole test suite generation. In *Proceedings of the 11th International Conference on Software Quality*, page 1, 2011.

[FAW07] Gordon Fraser, Bernhard K. Aichernig, and Franz Wotawa. Handling model changes: Regression testing and test-suite update with model-checkers. *MBT '07: Proceedings of the Third Workshop on Model Based Testing*, 190(2):33 – 46, 2007.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[FGX09] Chen Fu, Mark Grechanik, and Qing Xie. Inferring types of references to GUI objects in test scripts. In *International Conference on Software Testing Verification and Validation*, pages 1–10, 2009.

[FZ12] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[GAM09] Denaro Giovanni, Gorla Alessandra, and Pezeè Mauro. DaTeC: Contextual data flow testing of Java classes. In *Companion of the Proceedings of 31st International Conference on Software Engineering*, pages 421–422, 2009.

[GGJ+10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udita. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 225–234, 2010.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM: a research framework for the efficient execution of Java bytecode. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, page 3, 2001.

[GHS96] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. Program slicing-based regression testing techniques. *Software Testing Verification and Reliability*, 6(2):83–111, 1996.

[GJS08] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[God07] Patrice Godefroid. Compositional dynamic test generation. *ACM SIGPLAN Notices - Proceedings of the 2007 Annual Symposium on Principles of Programming Languages*, 42:47–54, January 2007.

[HGS93] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodololgy*, 2:270–285, 1993.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 312–326, 2001.

[HM10] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226 –247, 2010.

[HMF92] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick. Incremental testing of object-oriented class structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 68–80, 1992.

[HO08] Mary Jean Harrold and Alessandro Orso. Retesting software during development and maintenance. *Frontiers of Software Maintenance*, pages 99–108, 2008.

[HR90] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31 –36, 1990.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, 1994.

[HS05] Jason Osborne Hema Srikanth, Laurie Williams. System test case prioritization of new and regression test cases. In *Proceedings of the International Symposium on Empirical Software Engineering*, page 64Ð73, 2005.

[HZ93] James H. Hicinbothom and Wayne W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2, page 1042, 1993.

[IX08] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306, 2008.

[Jac90] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1990.

[JH03] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195 – 209, 2003.

[JLDM09] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 171–185, 2009.

[JLM11] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *ISSTA '11: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 133–143, 2011.

[JOX10a] Wei Jin, Alessandro Orso, and Tao Xie. Automated behavioral regression testing. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pages 137–146, 2010.

[JOX10b] Wei Jin, Alessandro Orso, and Tao Xie. BERT: a tool for behavioral regression testing. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 361–362, 2010.

[JS09] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92, 2009.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KTH05] Bogdan Korel, Luay H. Tahat, and Mark Harman. Test prioritization using system models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 559 – 568, 2005.

[Lev65] Vladimir I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[LHH07] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225 –237, 2007.

[LM90] Ursula Linnenkugel and Monika Müllerburg. Test data selection criteria for (software) integration testing. In *ISCI '90: Proceedings of the first international conference on systems integration on Systems integration*, pages 709–717, 1990.

[LMH10] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83:2379–2391, December 2010.

[LS92] Janusz Laski and Wojciech Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 282 –290, 1992.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[McM05] Phil McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield, 2005.

[MDR06] Henry Muccini, Marcio Dias, and Debra J. Richardsonn. Software architecture-based regression testing. *Journal of systems and software*, 79(10):1379–1396, 2006. Special Issue on 'Architecting Dependable Systems'.

[Mem04] Atif M. Memon. Using tasks to automate regression testing of GUIs. In *Proceedings of The IASTED International Conference on artificial intelligence and applications*, pages 477–482, 2004.

[Mem08] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions Software Engineering Methodology*, 18(2):1–36, 2008.

[Mir11] Mehdi Mirzaaghaei. Automatic test suite evolution. In *FSE '11: Proceedings of the 2011 Foundations of Software Engineering Conference*, pages 396–399, 2011.

[MM07] Scott McMaster and Atif Memon. Fault detection probability analysis for coverage-based test suite reduction. In *Proceedgins of the IEEE International Conference on Software Maintenance*, pages 335 –344, 2007.

[MNX05] Atif Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance*, 17(1):27–64, 2005.

[MP11] Mehdi Mirzaaghaei and Fabrizio Pastore. TestCareAssistant: Automatic repair of test case compilation errors. In *Proceedings of 6th Italian Workshop on Eclipse Technologies*, pages 90–101, 2011.

[MPP10] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. Automatically repairing test cases for evolving method declarations. In *ICSM '10: Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–5, 2010.

[MPP11] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. Algorithms for repairing test suites through parameters adaptation. Technical report, Faculty of Informatics, University of Lugano, Switzerland, 2011.

[MPP12] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. Supporting test suite evolution through test case adaptation. In *proceedings of the Fifth International Conference on Software Testing, Verification and Validation*, pages 231–240, 2012.

[MS03] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference and The 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, 2003.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, 2007.

[MT04] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 2004.

[Muc07] Henry Muccini. Using model differencing for architecture-level regression testing. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 59–66, 2007.

[NMS+11] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. Regression testing in the presence of non-code changes. In *Proceedings of Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 21 –30, 2011.

[NW70] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. *SIGSOFT Software Engineering Notes*, 29:241–251, 2004.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 87–96, 2008.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007.

[PMB+08] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26, 2008.

[PSO12] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *FSE '12: Proceedings of the 2012 Foundations of Software Engineering Conference*, 2012.

[PY07] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, Inc, 2007.

[REP+11] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *International Conference on Automated Software Engineering*, pages 23–32, 2011.

[RGJ06] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252, 2006.

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 169–184, 1994.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34 –43, 1998.

[RRST05] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for Java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, 2005.

[SB02] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[SCA+08] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *ASE '08: The 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[SRRE08] Michele Sama, Franco Raimondi, David S. Rosenblum, and Wolfgang Emmerich. Algorithms for efficient symbolic detection of faults in context-aware applications. In *Proceedings of 23rd*

IEEE/ACM International Conference on Automated Software Engineering Workshops, pages 1 –8, 2008.

[SS04] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with kaba. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, 2004.

[SSK03] Friedrich Steimann, Wolf Siberski, and Thomas Kühne. Towards the systematic use of interfaces in Java programming. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 13–17, 2003.

[ST02] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–106, 2002.

[SYC+04] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, 2004.

[Tah92] Abu-Bakr Mostafa Taha. *An approach to software fault localization and revalidation based on incremental data flow analysis*. PhD thesis, University of Florida, 1992.

[TBV07] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 97–107, 2007.

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .NET. In *TAP'08: Proceedings of the 2nd international conference on Tests and proofs*, pages 134–153, 2008.

[TdHTW10] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. DyGen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *TAP '10: Proceedings of the 4th international conference on Tests and proofs*, pages 77–93, 2010.

[Ton04] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[TXT+09] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: object-oriented unit-test generation via mining source code. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 193–202, 2009.

[TXT+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. *SIGPLAN Notices - Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 46(10):189–206, 2011.

[Utt05] Mark Utting. Position paper: Model-based testing. In *Verified Software: Theories, Tools, Experiments*, 2005.

[VF97] Filippos Vokolos and Phyllis Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proceedings of 3rd international conference on on Reliability, quality and safety of software-intensive systems*, pages 3–21, 1997.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java pathfinder. *SIGSOFT Software Engineering Notes*, 29:97–107, 2004.

[VPP06a] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, 2006.

[VPP06b] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, 2006.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999*

conference of the Centre for Advanced Studies on Collaborative research, page 13. IBM Press, 1999.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[WHLB97] Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium On Software Reliability Engineering*, pages 264 –274, 1997.

[WSKR06] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. TimeAware test suite prioritization. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12, 2006.

[WST09] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 173–182, 2009.

[XGF08] Qing Xie, Mark Grechanik, and Chen Fu. REST: A tool for reducing effort in script-based testing. In *IEEE International Conference on Software Maintenance*, pages 468–469, 2008.

[XKK+10] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *FSE '10: Proceedings of the 2010 Foundations of Software Engineering Conference*, pages 257–266, 2010.

[XN05] Tao Xie and David Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31:869–883, 2005.

[XR07] Guoqing Xu and Atanas Rountev. Regression test selection for AspectJ software. In *Proceedings of 29th International Conference on Software Engineering*, pages 65 –74, 2007.

[XR09] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *APSEC '09: Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, pages 406–413, 2009.

[XS06] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported an eclipse case study. In *ICSM'06: IEEE International Conference on Software Maintenance*, pages 458–468, 2006.

[ZRWS06] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley. Applying regression test selection for COTS-based applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 512–522, 2006.